Synthesizing Structured CAD Models with Equality Saturation and Inverse Transformations

Chandrakana Nandi
University of Washington
USA
cnandi@cs.washington.edu

Max Willsey
University of Washington
USA
mwillsey@cs.washington.edu

Adam Anderson
University of Washington
USA
adamand2@cs.washington.edu

James R. Wilcox
Certora
USA
james@certora.com

Eva Darulova
MPI-SWS
Germany
eva@mpi-sws.org

Dan Grossman
University of Washington
USA
djg@cs.washington.edu

Zachary Tatlock
University of Washington
USA
ztatlock@cs.washington.edu

Abstract

Recent program synthesis techniques help users customize CAD models (e.g., for 3D printing) by decompiling low-level triangle meshes to Constructive Solid Geometry (CSG) expressions. Without loops or functions, editing CSG can require many coordinated changes, and existing mesh decompilers use heuristics that can obfuscate high-level structure.

This paper proposes a second decompilation stage to robustly "shrink" unstructured CSG expressions into more editable programs with map and fold operators. We present Szalinski, a tool that uses Equality Saturation with semantics-preserving CAD rewrites to efficiently search for smaller equivalent programs. Szalinski relies on *inverse transformations*, a novel way for solvers to speculatively add equivalences to an E-graph. We qualitatively evaluate Szalinski in case studies, show how it composes with an existing mesh decompiler, and demonstrate that Szalinski can shrink large models in seconds.

CCS Concepts: • **Software and its engineering** → **Compilers**; **Domain specific languages**; **Software reverse engineering**; • **Theory of computation** → **Program semantics**; • **Computing methodologies** → **Parametric curve and surface models**.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PLDI '20, June 15–20, 2020, London, UK

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-7613-6/20/06…$15.00
https://doi.org/10.1145/3385412.3386012

Keywords: Equality Saturation, Program Synthesis, Decompilation, Computer-Aided Design

ACM Reference Format:
Chandrakana Nandi, Max Willsey, Adam Anderson, James R. Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. 2020. Synthesizing Structured CAD Models with Equality Saturation and Inverse Transformations. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3385412.3386012

1 Introduction

The programming languages and machine learning communities have developed techniques to decompile Computer-Aided Design (CAD) models from low-level numerical representations to Constructive Solid Geometry (CSG) expressions [11, 13, 14, 21, 30, 31, 36]. These techniques aim to help users modify designs shared in online repositories [1, 15, 35].

Recent program synthesis results [11, 21] decompile *meshes*, 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.

Existing mesh decompilers synthesize *flat* output: CSG has no loops or functions (Figure 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 [11, 21], so past tools rely on heuristics which cause them to exhibit two challenging features: **(C1)** synthesize equivalent but dissimilar CSG expressions for the same feature repeated under different transformations, and **(C2)** arbitrarily order CSG

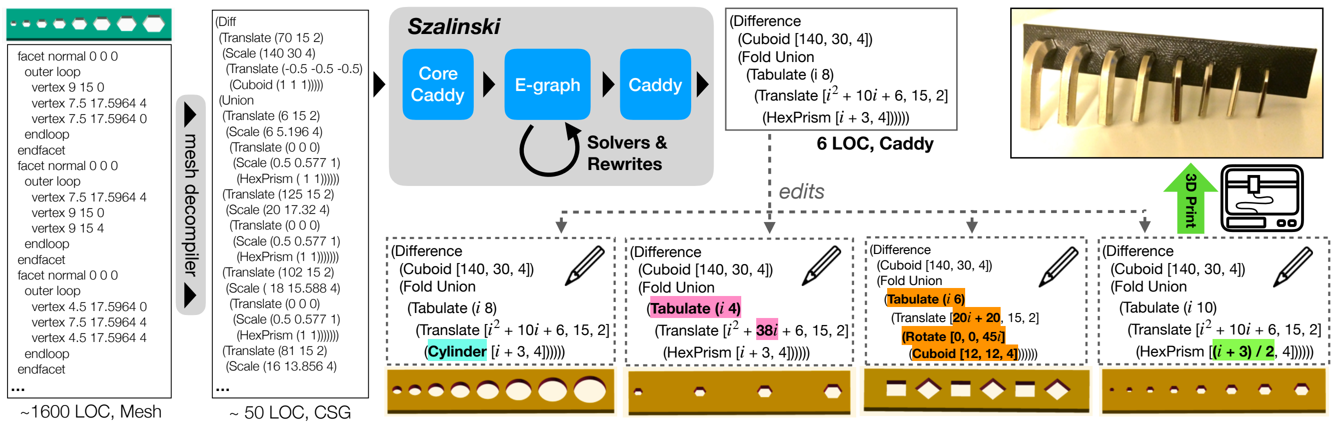


Figure 1. Existing mesh decompilers turn triangle meshes into CSG expressions. Szalinski robustly synthesizes smaller, structured Caddy programs from CSG expressions. This can ease customization by simplifying edits: small, mostly local changes yield usefully different models. The photo shows the 3D printed hex wrench holder after customizing hole sizes.

subexpressions. These two features, **(C1)** and **(C2)** obfuscate high-level structure latent in synthesized CSG.

This paper 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 present *Szalinski*[1] (Figure 1), a tool which combines semantics-preserving rewrites with simple solvers to synthesize structured CAD programs in a language called *Caddy*.

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 Saturation [34] suggests that Equality Graphs (E-graphs) [22]—an efficient data structure underlying SMT solvers [8, 10] and program optimizers [16, 33, 34, 42]—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* [3, 6, 17]. 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 in Szalinski we present *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.

To summarize, the contributions of this paper include:

- *Szalinski*, a tool that takes a flat CSG expression as input and synthesizes a smaller equivalent program in *Caddy*, a language that extends CSG with map and fold operators for expressing repetition.
- *Inverse transformations*, a new technique for interfacing simple-yet-effective structure finding solvers with E-graphs. The technique is not CAD-specific, but is particularly useful for reordering CAD operations.
- A case study composing Szalinski with a recent mesh decompiler [21] to synthesize smaller CAD models.
- A large scale evaluation demonstrating the performance and scalability of Szalinski on models downloaded from a popular online repository [35].

This paper proceeds gradually, first introducing Caddy and a running example (Section 2). Szalinski primarily exploits opportunities to "reroll loops" (Section 3). Finding such opportunities is challenging due to variations in mesh decompiler output **(C1)**, so Szalinski uses E-graphs to implement a robust CAD rewrite system (Section 4). Finding the right CAD reordering is crucial to expose high-level structure **(C2)**, but difficult with rewrites alone due to AC-matching. Solvers in Szalinski propagate profitable reorderings through the E-graph by unifying order-inequivalent expressions annotated with inverse transformations (Section 5).

We developed a library of 65 CAD rewrites and prototyped Szalinski in 3,000 lines of Rust (Section 6). Section 7 shows how composing Szalinski with an existing mesh decompiler [21] qualitatively improves editability (sketched in

[1]The protagonist in the hit movie Honey I Shrunk the Kids was named Dr. Szalinski. Our work shrinks CADs rather than kids.

Figure 1) and describes an evaluation of Szalinski's performance and correctness on real-world CAD models downloaded from Thingiverse. Section 8 briefly surveys the most relevant related work and Section 9 concludes.

2 Caddy and Second Stage Decompilation

The *Caddy* language (Figure 2) provides map- and fold-like functional list operators to express repetitive structure in CAD models, as well as a *Core Caddy* fragment that corresponds directly to CSG. The Caddy semantics fully unroll a program's functional list operators to produce a Core Caddy (CSG) expression. Szalinski "goes the other way," decompiling a Core Caddy expression to a Caddy 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.

2.1 Core Caddy, Caddy, Equivalence

Core Caddy includes various primitives parametrized by dimensions— cuboids parametrized by side length, spheres by radius, cylinders and hexagonal prisms by height and radius, etc. Caddy 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 Caddy or CSG) captures a common pattern in models relying on translations in spherical rather than Cartesian coordinates.

Figure 3 gives semantics for the functional list operators Caddy provides on top of Core Caddy. Tabulate takes pairs of variables and positive integers $(x_1\ b_1) \dots (x_n\ b_n)$ as well as a Caddy expression e, and returns the list of length Πb_i generated by n nested loops evaluating e over the variables $x_1 \dots x_n$ up to the bounds $b_1 \dots b_n$:

$$(\text{List}\quad e[0/x_1]\dots[0/x_n]\quad \dots \quad e[b_1-1/x_1]\dots[b_n-1/x_n])$$

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]))
```

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 Caddy 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)))
```

Caddy programs are equivalent iff they evaluate to equivalent Core Caddy programs. By design, Core Caddy directly corresponds to CSG, whose semantics is given in prior work [21, 27, 31]. Section 7 describes practically testing Caddy equivalence by evaluating programs to Core Caddy, compiling them to meshes, and comparing Hausdorff distances.[4]

op	::=	+ | - | × | / num ::= ℝ | ⟨*var*⟩ | ⟨*num*⟩ ⟨*op*⟩ ⟨*num*⟩
vec2	::=	[⟨*num*⟩, ⟨*num*⟩] vec3 ::= [⟨*num*⟩, ⟨*num*⟩, ⟨*num*⟩]
affine	::=	Translate | Rotate | Scale | **TranslateSpherical**
binop	::=	Union | Difference | Intersection
cad	::=	(Cuboid ⟨*vec3*⟩) | (Sphere ⟨*num*⟩)
	|	(Cylinder ⟨*vec2*⟩) | (HexPrism ⟨*vec2*⟩) | …
	|	(⟨*affine*⟩ ⟨*vec3*⟩ ⟨*cad*⟩)
	|	(⟨*binop*⟩ ⟨*cad*⟩ ⟨*cad*⟩)
	|	**(Fold ⟨*binop*⟩ ⟨*cad-list*⟩)**
cad-list	::=	(List ⟨*cad*⟩+)
	|	(Concat ⟨*cad-list*⟩+)
	|	(Tabulate (⟨*var*⟩ $\mathbb{Z}^+$)+ ⟨*cad*⟩)
	|	(Map2 ⟨*affine*⟩ ⟨*vec3-list*⟩ ⟨*cad-list*⟩)
vec3-list	::=	(List ⟨*vec3*⟩+)
	|	(Concat ⟨*vec3-list*⟩+)
	|	(Tabulate (⟨*var*⟩ $\mathbb{Z}^+$)+ ⟨*vec3*⟩)

Figure 2. Caddy syntax. The Core Caddy (CSG) subset omits variables, list forms (those using **Fold**), and **TranslateSpherical**.

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

$$\frac{e \Rightarrow\ (\text{List}\ (\text{List}\ v_{1,1}\ v_{1,2} \dots)\ (\text{List}\ v_{2,1}\ v_{2,2} \dots) \dots)}{(\text{Concat}\ e) \Rightarrow (\text{List}\ v_{1,1}\ v_{1,2} \dots v_{2,1}\ v_{2,2} \dots)}$$

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

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

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

Figure 3. Big step semantics reducing well-formed Caddy programs to Core Caddy 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.

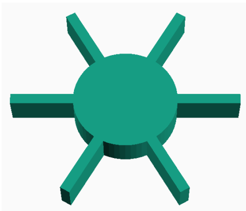
2.2 A Running Example for Shrinking Caddy

Figure 4a shows a simple CAD model of a ship's wheel and Figure 4b shows the corresponding desired Caddy output

[2] We use 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.

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



(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) Caddy 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 Caddy expression that exposes structure

```
(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 Core Caddy expression that obfuscates structure

Figure 4. (a) CAD model for a ship's wheel. (b) Caddy features like Tabulate express repeated design components. Such repetition can be obvious in Core Caddy (c), but existing mesh decompilers obfuscate structure (d).

from Szalinski. Figure 4b reifies repetitive structure: making a change to all the spokes only requires a single edit instead of six coordinated modifications in different locations.

When repetitive structure is easily exposed, as in the ideal Core Caddy of Figure 4c, solvers can infer the arithmetic function relating instances of repeated design components. Section 3 describes Szalinski's rewrite-driven approach to infer such functions and shrink programs by rerolling loops.

In practice, given a mesh representing Figure 4a, mesh decompilers can generate CSG expressions equivalent to Figure 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 4c to 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 and 5 walk through progressively more challenging variants of Core Caddy inputs for the ship's wheel to illustrate how Szalinski uses E-graphs and inverse transformations to robustly handle such variation.

Binop Fold

(*binop* c_1 c_2 ...) $\rightsquigarrow$ (Fold *binop* (List c_1 c_2 ...))

Structure Finding

(List (*aff* p_1 c_1) (*aff* p_2 c_2) ...) $\rightsquigarrow$ (Map2 *aff* (List p_1 p_2 ...) (List c_1 c_2 ...))

Repeat

(List a a a ... n times) $\rightsquigarrow$ (Repeat n a)

List Solve (single loop)

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

Repeat over Map2

(Map2 *aff* (Repeat n p) (Repeat n c)) $\rightsquigarrow$ (Repeat n (*aff* p c))

Tabulate over Map2 where $b = \Pi b_i$

(Map2 *aff* (Tabulate (x_1 b_1) ... p) (Tabulate (x_1 b_1) ... c))
$\rightsquigarrow$ (Tabulate (x_1 b_1) ... (*aff* p c))

(Map2 *aff* (Tabulate (x_1 b_1) ... p) (Repeat b c))
$\rightsquigarrow$ (Tabulate (x_1 b_1) ... (*aff* p c))

(Map2 *aff* (Repeat b p) (Tabulate (x_1 b_1) ... c))
$\rightsquigarrow$ (Tabulate (x_1 b_1) ... (*aff* p c))

Figure 5. Rewrite rules for loop rerolling

3 Shrinking Caddy by Rerolling Loops

Szalinski shrinks repetitive Caddy programs by "rerolling loops". First, rewrites *find structure* by separating affine operators from their parameters and CAD arguments under Map2s. 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 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 Caddy input from Figure 4c.

3.1 Finding Structure: A Bird's-Eye View

Applying **Binop Fold** to the inner Union in Figure 4c produces:

```
(Union (Union (Union ...                 ▶   (Fold Union (List
 (Rotate [0, 0, 0]   cad1)                    (Rotate [0, 0, 0]   cad1)
 (Rotate [0, 0, 60]  cad2)                    (Rotate [0, 0, 60]  cad2)
 (Rotate [0, 0, 120] cad3) ...)               (Rotate [0, 0, 120] cad3) ...))
```

A structure finder (detailed in Section 4) 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]    cad1)
  (Rotate [0, 0, 60]   cad2)
  (Rotate [0, 0, 120]  cad3) ...)))
▶
(Fold Union
  (Map2 Rotate
    (List [0, 0, 0] [0, 0, 60] [0, 0, 120] ...)
    (List cad1 cad2 cad3 ...)))
```

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])))))
```

3.2 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 $\ell = (\text{List}\ [x_1, y_1, z_1] \ldots [x_n, y_n, z_n])$, 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 [11] like RANSAC [28].

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

In this example, the solvers relied on their input arriving in just the right order. Section 5 shows how inverse transformations allow solvers to *reorder* their input to infer better Tabulates while preserving equivalence.

3.3 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 4c:

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

This section illustrated Szalinski's core strategy: shrinking Caddy by rerolling loops. However, the example relied on a specific rewrite order and Figure 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 E-graphs and CAD Equality Saturation

Rewrites to shrink Caddy by rerolling loops must be applied in just the right order to programs that already make structure apparent as in Figure 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 [34] demonstrates how E-graphs [22] 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 Caddy programs.

4.1 Rewrite Phase Ordering: What, When, Where

A slightly perturbed Caddy 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 3 now breaks: Szalinski must interleave its search with additional CAD rewrites (Figure 6) to expose the repeated affine transformations as in Figure 4c. This *phase ordering problem* [34, 38] 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 [34] 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 5 and 6 to an E-graph and using a structure finding heuristic (Section 4.4), Szalinski's loop rerolling strategy can robustly handle variations in how mesh decompilers synthesize affine operators.

4.2 E-graph Background

An E-graph is a set of *eclasses*, and each eclass is a set of equivalent *enodes*. An enode is an operator (Translate, Union, 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.

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 6. Selected CAD identities. Bidirectional arrows indicates Szalinski has a rule for each direction.

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

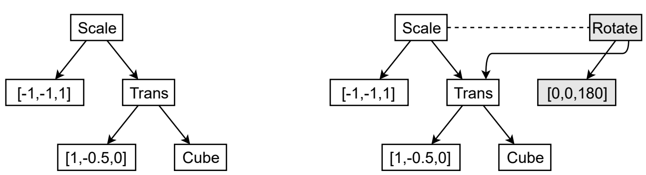


Figure 7. E-graph before and after a CAD rewrite. Boxes represent enodes and dashed edges indicate equivalence (membership in the same eclass). Directed solid edges connect enodes to their child eclasses. Both the original and transformed programs are represented in the resulting E-graph.

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 ensures enodes are never duplicated in an E-graph. This sharing compactly represents many equivalent expressions.

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. Figure 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.

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 $\phi(b)$. Rewrites only expand the E-graph, all previous expressions are still represented.

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 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.

```
def Szalinski(csg : core_caddy):
    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 8. Equality Saturation for Caddy in Szalinski

4.3 Equality Saturation in Szalinski

Szalinski implements Equality Saturation [34] for Caddy (Figure 8). First, an E-graph is created from the input Core Caddy 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 Caddy program represented by the initial Core Caddy 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 [24, 34], but we leave further exploration of CAD cost functions in Szalinski to future work.

4.4 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 3.1 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 $e_1, e_2, \ldots, e_n$, the structure finder aims to extract Map2s that remove one level of structure. However, due to rules like Affine Combination from Figure 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 Caddy 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 4.1. After applying the two Rotate identities from Figure 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 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.

```
(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 9. Section 3 and 4 techniques find the "Rotate then Translate" structure from the realistic Figure 4d. Without inverse transformations, loop rerolling now gets stuck.

5 Inverse Transformations

E-graphs and CAD rewrites allow Szalinski to expose repetitive structure and reroll loops even when a Core Caddy 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 Caddy 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 9 shows how far CAD rewrites combined with techniques from previous sections get for the Figure 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 [3] (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 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.

[5] We can report that AC-matching is a problem both in theory and practice.

permutation::= ⟨ n, n, ... ⟩ partitioning ::= ⟨ n, n, ... ⟩

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

Figure 10. Syntax of Extended Caddy.

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).

5.1 Extended Caddy

Extended Caddy (Figure 10 and 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 Caddy 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 use 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).

5.2 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 4d). Starting from Figure 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 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 9, the partitioner will split the list into:

```
(Fold Union
  (Unpart ⟨1, 6⟩
    ( List (Cylinder [1, 5]))
```

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

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

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

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

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

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

Figure 11. Big step semantics for Extended Caddy.

```
( 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 3.2 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 Caddy. 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 12. The **Map2 Unsort Params** rewrite applies to our 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 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 Caddy output (Figure 4b).

5.3 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 [19]. 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.

5.4 Inverse Transformations, Broadly

This section and our evaluation show that inverse transformations are effective for shrinking Caddy 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.

Map2 Unsort Params - *cads* rule analogous

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

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

Unpart to Concat

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

Unspherical Trans

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

Figure 12. Representative set of rewrite rules for propagation and elimination of inverse transformations.

6 Implementation

Szalinski is implemented in 3000 lines of Rust and uses egg [41], an open source E-graph library. Table 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. The most of Szalinski's 65 rewrites are syntactic and compactly expressed, and the remainder either call out to the solvers or manipulate lists. Szalinski is publicly available at *https://github.com/uwplse/szalinski.git*.

Table 1. Approximate LOC breakdown of Szalinski

Caddy	Rewrites	Solvers	Main loop	Validation
300	900	400	300	1100

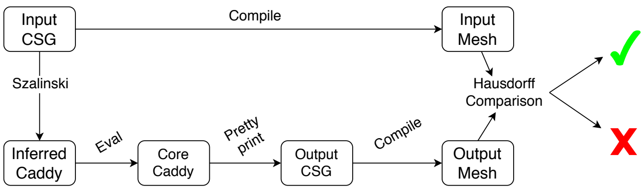


Figure 13. The Szalinski tool. The simplification process outputs a parameterized program in the Caddy language. The validation step evaluates Szalinski's output to Core Caddy, pretty prints it to CSG and uses an open source CAD compiler to generate a mesh. The input to Szalinski is also compiled to a mesh. The two meshes are then compared using Hausdorff distance.

Correctness. To validate Szalinski's correctness, we test that the initial and final Caddy programs compile to similar meshes (Figure 13). Szalinski first evaluates a Caddy program back to a flat Core Caddy program which is then pretty printed to a CSG program. We use the open source OpenSCAD [23] tool to compile the CSGs to triangular meshes. We then use the CGAL [4] library to compute the Hausdorff distance [11, 20] 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).

7 Evaluation

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

- *End-to-End.* (Section 7.1) Does Szalinski compose with prior mesh decompilation tools and find parametrizable programs from the flat CSG expressions generated by the latter?
- *Scalability.* (Section 7.2) Does Szalinski scale to large flat CSGs? How fast can it find equivalent smaller Caddy programs?
- *Sensitivity analysis.* How do the different components of Szalinski, in particular CAD rewrites and inverse transformations, affect its results?

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

7.1 End-to-End Experiments

To evaluate the composability of Szalinski with mesh decompilation tools, we ran Szalinski on flat CSGs generated by the Reincarnate [21] mesh decompiler. This required 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 [35] and ran Reincarnate on their mesh files to get flat CSGs and converted those to Core Caddy.

Table 2. End-to-end evaluation of Szalinski on the results of Reincarnate [21]. 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 Caddy 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

Given the Core Caddy inputs, Szalinski synthesizes Caddy programs (Figure 13). We compared the parametrized programs synthesized by Szalinski from Reincarnate's output with manually written parametrized programs in OpenSCAD (column 1 in Table 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 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 Caddy and the output Caddy. 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 Caddy programs.

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

7.2 Large Scale Evaluation on Thingiverse Models

Mesh decompilation tools have limitations. Reincarnate for example, works mainly on shapes without rounded corners and edges. Therefore, in order to evaluate Szalinski further, we performed a larger scale evaluation on models from Thingiverse [35], 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 in Section 1), and run Szalinski on these flat CSGs, (2) to analyze the scalability,

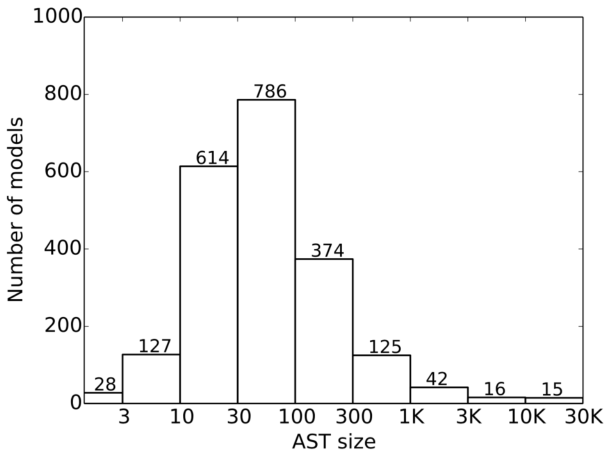


Figure 14. Summary of input AST size for Szalinski's large scale evaluation.

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 [7] are intended to be editable, and are therefore more likely to be accompanied with higher-level programmatic representation. Our 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 Caddy, 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 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. We found 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 our verifier (Section 6) uses the OpenSCAD compiler, we capped the precision of all primitives to 25.

Results. Figure 15 shows our results with a 60 second timeout. We refer to the baseline result (leftmost) as 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 with the by comparing the meshes. All Hausdorff distances were under 0.01, except for 148 cases where CGAL failed to compute the distance and we visually compared the meshes.

7.3 Case Studies and Editability

This section discusses three models from the end-to-end evaluation in Section 7.1 (a fourth is illustrated in Figure 1) and three models from the large scale evaluation in Section 7.2. The goal is to highlight some edits made easily possible by Szalinski, which in the flat CSG (and mesh) are nearly impossible. Figure 16 shows a rendering of these models and the parametrized Caddy program found by Szalinski. We discuss three categories of edits.

Adding or removing components: consider the gear shown in Figure 16. Changing the tooth count in a flat CSG version of this model requires manually computing the position of every teeth and ensuring that the spacing between them is still equal. The Caddy program synthesized by Szalinski makes this modification trivial—it exposes a function ($6 \times i$) for Rotate and the number of teeth (in the 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 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 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 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 16. This model can be easily customized in the Caddy program to adjust the size of the slots. The Caddy 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 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

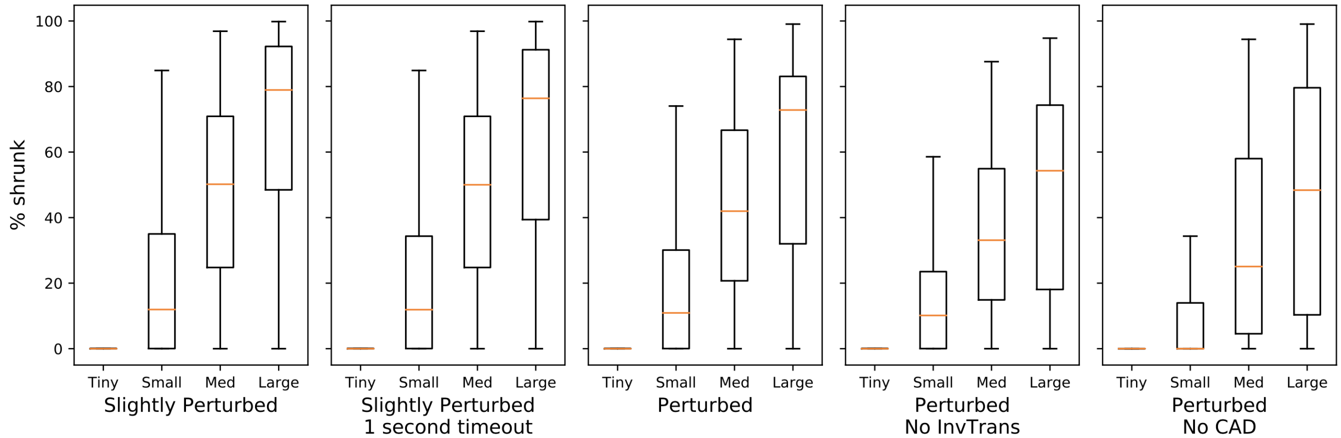
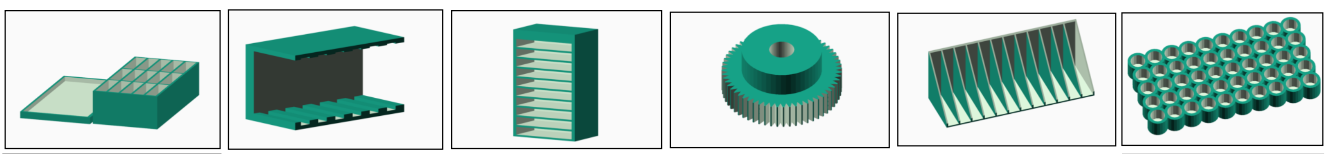


Figure 15. Result of running Szalinski on 2,127 Thingiverse examples. Models are grouped by AST size of initial Core Caddy input: 769 were tiny (AST size < 30), 786 small (30 < size < 100), 374 medium (100 < size < 300), and 198 large (300 < size).



```
(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 16. The first three are examples of end-to-end evaluation where Szalinski ran on the flat CSG output of a mesh decompiler [21]. 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.

many parameters for multiple components in the models. Szalinski makes these modifications much easier by exposing different design parameters.

7.4 Limitations

Some mesh decompilation tools like InverseCSG synthesize flat CSG programs using enumerative synthesis and random sampling based algorithms like RANSAC [11]. Inferring structure from the output generated by these tools requires equivalence under context using geometric reasoning that our 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)` ⇝ `(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.

8 Related Work

E-graph based Deductive Program Synthesis. E-graphs have been used extensively in superoptimizers [2, 16, 33, 34], and SMT solvers [8–10, 37]. Szalinski's core algorithm is a generalized version of equality saturation [34]. Integrating linear solvers with compiler optimizers has a long history with tools like Omega Calculator [25, 26]. Our approach of using syntactic rewrites and an arithmetic function solver to modify the E-graph can be considered similar to Simplify [10] 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. Nandi et al. [21] and Du et al. [11] have developed tools that can decompile low-level polygon meshes to flat CSGs. These tools use program synthesis together with domain specific computational geometric algorithms to discover structure in the meshes. CSGNet [30] uses machine learning to generate flat CSG programs for 2D and 3D shapes. Shape2Prog [36] uses machine learning to infer programs from voxel-based 3D models. They use LSTMs to infer programs with loops. We ran Szalinski on the flat CSGs from both CSGNet and Shape2Prog— since their program lengths are very small (AST depth < 7), they are not good candidates for design parameter inference. Szalinski however did find some structure in these program and generated correct outputs. Ellis et al. [13] 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 [32] 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 [13] 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 [12] 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 [18] 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. [29] developed algorithms for optimizing parametric CAD models using interpolation methods. While their approach can optimize parameters, it does not automatically infer maps and folds from flat CSG inputs.

9 Conclusion

This paper 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 *Caddy* 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 [39, 40].

To the best of our knowledge, Szalinski is the first tool of its kind. We performed an early survey of 2,127 real-world CAD models from Thingiverse. Our evaluation shows that Szalinski can dramatically shrink many CAD models in seconds.

In future work, we are excited to explore richer rewrites for contextual equivalence (Section 7.4), more expressive cost functions for capturing richer notions of editability, and connections to interactive CAD editing using direct manipulation tools like Sketch-n-Sketch [5].

Acknowledgments

We thank René Just for valuable suggestions on data visualization, and other members of the UWPLSE lab for feedback on early drafts. We also thank our shepherd, Mukund Raghotaman, and the anonymous reviewers for their guidance while preparing the final revision of this paper. This material is based upon work supported by the National Science Foundation under Grant No. CCF-1813166.

References

[1] Celena Alcock, Nathaniel Hudson, and Parmit K. Chilana. 2016. Barriers to Using, Customizing, and Printing 3D Designs on Thingiverse. In *Proceedings of the 19th International Conference on Supporting Group Work (GROUP '16)*. ACM, New York, NY, USA, 195–199. https://doi.org/10.1145/2957276.2957301

[2] Sorav Bansal and Alex Aiken. 2008. Binary Translation Using Peephole Superoptimizers. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, USA, 177–192. http://dl.acm.org/citation.cfm?id=1855741.1855754

[3] Walid Belkhir and Alain Giorgetti. 2012. Lazy AC-Pattern Matching for Rewriting. *Electronic Proceedings in Theoretical Computer Science* 82 (Apr 2012), 37–51. https://doi.org/10.4204/eptcs.82.3

[4] CGAL. 2018. CGAL. https://www.cgal.org.

[5] Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. 2016. Programmatic and Direct Manipulation, Together at Last. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 341–354. https://doi.org/10.1145/2908080.2908103

[6] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. 2007. *All About Maude - a High-performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic.* Springer-Verlag, Berlin, Heidelberg.

[7] Customizable. 2019. Thingiverse Customizable. https://www.thingiverse.com/customizable.

[8] Leonardo De Moura and Nikolaj Bjørner. 2008. 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)*. Springer-Verlag, Berlin, Heidelberg, 337–340. http://dl.acm.org/citation.cfm?id=1792734.1792766

[9] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description). In *Automated Deduction - CADE-25*, Amy P. Felty and Aart Middeldorp (Eds.). Springer International Publishing, Cham, 378–388.

[10] David Detlefs, Greg Nelson, and James B. Saxe. 2005. Simplify: A Theorem Prover for Program Checking. *J. ACM* 52, 3 (May 2005), 365–473. https://doi.org/10.1145/1066100.1066102

[11] Tao Du, Jeevana Priya Inala, Yewen Pu, Andrew Spielberg, Adriana Schulz, Daniela Rus, Armando Solar-Lezama, and Wojciech Matusik. 2018. InverseCSG: automatic conversion of 3D models to CSG trees. 1–16. https://doi.org/10.1145/3272127.3275006

[12] Kevin Ellis, Maxwell I. Nye, Yewen Pu, Felix Sosa, Joshua B. Tenenbaum, and Armando Solar-Lezama. 2019. Write, Execute, Assess: Program Synthesis with a REPL. In *NeurIPS*.

[13] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Joshua B. Tenenbaum. 2018. Learning to Infer Graphics Programs from Hand-Drawn Images. https://openreview.net/forum?id=H1DJFybC-

[14] Markus Friedrich, Pierre-Alain Fayolle, Thomas Gabor, and Claudia Linnhoff-Popien. 2019. Optimizing Evolutionary CSG Tree Extraction. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '19)*. ACM, New York, NY, USA, 1183–1191. https://doi.org/10.1145/3321707.3321771

[15] Nathaniel Hudson, Celena Alcock, and Parmit K. Chilana. 2016. 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)*. ACM, New York, NY, USA, 384–396. https://doi.org/10.1145/2858036.2858266

[16] Rajeev Joshi, Greg Nelson, and Keith Randall. 2002. Denali: A Goal-directed Superoptimizer. *SIGPLAN Not.* 37, 5 (May 2002), 304–314. https://doi.org/10.1145/543552.512566

[17] Hélène Kirchner and Pierre-Etienne Moreau. 2001. Promoting Rewriting to a Programming Language: A Compiler for Non-deterministic Rewrite Programs in Associative-commutative Theories. *J. Funct. Program.* 11, 2 (March 2001), 207–251. http://dl.acm.org/citation.cfm?id=968486.968488

[18] Niloy J. Mitra, Mark Pauly, Michael Wand, and Duygu Ceylan. 2013. Symmetry in 3D Geometry: Extraction and Applications. *Comput. Graph. Forum* 32, 6 (Sept. 2013), 1–23. https://doi.org/10.1111/cgf.12010

[19] P.H. Moon and D.E. Spencer. 1988. *Field theory handbook: including coordinate systems, differential equations, and their solutions.* Springer-Verlag. https://books.google.com/books?id=EDnvAAAAMAAJ

[20] James R Munkers. 2000. Topology.

[21] Chandrakana Nandi, James R. Wilcox, Pavel Panchekha, Taylor Blau, Dan Grossman, and Zachary Tatlock. 2018. Functional Programming for Compiling and Decompiling Computer-aided Design. *Proc. ACM Program. Lang.* 2, ICFP, Article 99 (July 2018), 31 pages. https://doi.org/10.1145/3236794

[22] Charles Gregory Nelson. 1980. *Techniques for Program Verification.* Ph.D. Dissertation. Stanford, CA, USA. AAI8011683.

[23] OpenScad. 2019. OpenScad. The Programmers Solid 3D CAD Modeller. http://www.openscad.org/.

[24] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. 2015. Automatically Improving Accuracy for Floating Point Expressions. *SIGPLAN Not.* 50, 6 (June 2015), 1–11. https://doi.org/10.1145/2813885.2737959

[25] William Pugh. 1991. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings Supercomputing '91, Albuquerque, NM, USA, November 18-22, 1991.* 4–13. https://doi.org/10.1145/125826.125848

[26] William Pugh and David Wonnacott. 1992. 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.* 140–151. https://doi.org/10.1145/143095.143129

[27] C. Ronse. 1990. Regular open or closed sets.

[28] R. Schnabel, R. Wahl, and R. Klein. 2007. Efficient RANSAC for Point-Cloud Shape Detection. *Computer Graphics Forum* (2007). https://doi.org/10.1111/j.1467-8659.2007.01016.x

[29] Adriana Schulz, Jie Xu, Bo Zhu, Changxi Zheng, Eitan Grinspun, and Wojciech Matusik. 2017. Interactive Design Space Exploration and Optimization for CAD Models. *ACM Trans. Graph.* 36, 4, Article 157 (July 2017), 14 pages. https://doi.org/10.1145/3072959.3073688

[30] Gopal Sharma, Rishabh Goyal, Difan Liu, Evangelos Kalogerakis, and Subhransu Maji. 2017. CSGNet: Neural Shape Parser for Constructive Solid Geometry. *CoRR* abs/1712.08290 (2017). arXiv:1712.08290 http://arxiv.org/abs/1712.08290

[31] Benjamin Sherman, Jesse Michel, and Michael Carbin. 2019. Sound and Robust Solid Modeling via Exact Real Arithmetic and Continuity. *Proc. ACM Program. Lang.* 3, ICFP, Article 99 (July 2019), 29 pages. https://doi.org/10.1145/3341703

[32] Armando Solar-Lezama. 2008. *Program Synthesis by Sketching.* Ph.D. Dissertation. Berkeley, CA, USA. Advisor(s) Bodik, Rastislav. AAI3353225.

[33] Michael Stepp, Ross Tate, and Sorin Lerner. 2011. Equality-Based Translation Validator for LLVM. In *Computer Aided Verification*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 737–742.

[34] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality Saturation: A New Approach to Optimization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '09)*. ACM, New York, NY, USA, 264–276. https://doi.org/10.1145/1480881.1480915

[35] Thingiverse. 2019. Thingiverse. https://www.thingiverse.com/.

[36] Yonglong Tian, Andrew Luo, Xingyuan Sun, Kevin Ellis, William T. Freeman, Joshua B. Tenenbaum, and Jiajun Wu. 2019. Learning to Infer and Execute 3D Shape Programs. In *International Conference on Learning Representations.*

[37] Emina Torlak and Rastislav Bodik. 2014. A Lightweight Symbolic Virtual Machine for Solver-aided Host Languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 530–541. https://doi.org/10.1145/2594291.2594340

[38] Sid-Ahmed-Ali Touati and Denis Barthou. 2006. On the Decidability of Phase Ordering Problem in Optimizing Compilation. In *Proceedings of the 3rd Conference on Computing Frontiers (CF '06)*. ACM, New York, NY, USA, 147–156. https://doi.org/10.1145/1128022.1128042

[39] Jean-Baptiste Tristan and Xavier Leroy. 2008. Formal Verification of Translation Validators: A Case Study on Instruction Scheduling Optimizations. *SIGPLAN Not.* 43, 1 (Jan. 2008), 17–27. https://doi.org/10.1145/1328897.1328444

[40] Jean-Baptiste Tristan and Xavier Leroy. 2009. Verified Validation of Lazy Code Motion. *SIGPLAN Not.* 44, 6 (June 2009), 316–326. https://doi.org/10.1145/1543135.1542512

[41] Max Willsey, Yisu Remy Wang, Oliver Flatt, Chandrakana Nandi, Pavel Panchekha, and Zachary Tatlock. 2020. egg: Easy, Efficient, and Extensible E-graphs. arXiv:cs.PL/2004.03082

[42] Chenming Wu, Haisen Zhao, Chandrakana Nandi, Jeffrey I. Lipton, Zachary Tatlock, and Adriana Schulz. 2019. Carpentry Compiler. *ACM Trans. Graph.* 38, 6, Article 195 (Nov. 2019), 14 pages. https://doi.org/10.1145/3355089.3356518