Functional Programming for Compiling and Decompiling Computer-Aided Design

CHANDRAKANA NANDI, University of Washington, USA
JAMES R. WILCOX, University of Washington, USA
PAVEL PANCHEKHA, University of Washington, USA
TAYLOR BLAU, University of Washington, USA
DAN GROSSMAN, University of Washington, USA
ZACHARY TATLOCK, University of Washington, USA

Desktop-manufacturing techniques like 3D printing are increasingly popular because they reduce the cost and complexity of producing customized objects on demand. Unfortunately, the vibrant communities of early adopters, often referred to as "makers," are not well-served by currently available software pipelines. Users today must compose idiosyncratic sequences of tools which are typically superposed variants of proprietary software originally designed for expert specialists.

This paper proposes fundamental programming-languages techniques to bring improved rigor, reduced complexity, and new functionality to the computer-aided design (CAD) software pipeline for applications like 3D-printing. Compositionality, denotational semantics, compiler correctness, and program synthesis all play key roles in our approach, starting from the perspective that solid geometry is a programming language.

Specifically, we define a purely functional language for CAD called λCAD and a polygon surface-mesh intermediate representation. We then define denotational semantics of both languages to 3D solids and a compiler from CAD to mesh accompanied by a proof of semantics preservation. We illustrate the utility of this foundation by developing a novel synthesis algorithm based on evaluation contexts to "reverse compile" difficult-to-edit meshes downloaded from online maker communities back to more-editable CAD programs. All our prototypes have been implemented in OCaml to enable further exploration of functional programming for desktop manufacturing.

CCS Concepts: • **Theory of computation → Program semantics**; • **Software and its engineering → Compilers**;

Additional Key Words and Phrases: language design, denotational semantics, program synthesis, 3D printing

ACM Reference Format:
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 (September 2018), 31 pages. https://doi.org/10.1145/3236794

1 INTRODUCTION

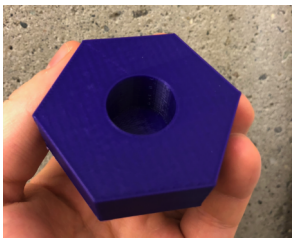
Democratized computer-aided manufacturing has made available—at modest cost—design and fabrication capabilities that were previously reserved for large-scale commercial applications.

Authors' addresses: Chandrakana Nandi, University of Washington, USA, cnandi@cs.washington.edu; James R. Wilcox, University of Washington, USA, jrw12@cs.washington.edu; Pavel Panchekha, University of Washington, USA, pavpan@cs.washington.edu; Taylor Blau, University of Washington, USA, ttaylorr@cs.washington.edu; Dan Grossman, University of Washington, USA, djg@cs.washington.edu; Zachary Tatlock, University of Washington, USA, ztatlock@cs.washington.edu.

2475-1421/2018/9-ART99
https://doi.org/10.1145/3236794

```
⟦ Diff (
    Scale (2.5, 2.5, 1) (
      Cylinder(6)
    )
    Scale (1, 1, 0.9) (
      Translate (0, 0, 0.5) (
        Cylinder(50)
      )
    )
  ) ⟧
```

=



A computer-aided design program denotes a geometric object.

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 paper takes preliminary steps toward addressing 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 in this paper—the first synthesis algorithm

to our knowledge that converts meshes back into CAD programs, which we view as a reverse compilation task. Given a surface mesh, our algorithm finds a CAD program, which, when compiled, gives that mesh.

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 [GrabCAD 2018; Thingiverse 2018b] 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. Our algorithm 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 have implemented the synthesis algorithm in a prototype tool called ReIncarnate. We detail three case studies that demonstrate our approach on samples selected from Thingiverse. They cover three popular applications of desktop 3D printing: machine tools, household objects, and aesthetic items designed by hobbyists. We give examples of modifications that are intractable to make in mesh models but become simple at the CAD level.

Our implementation of the compiler and synthesis prototypes consists of about 20000 LOC of OCaml[1]. As Section 4.3 describes, we leverage several features of OCaml to achieve simplicity and modularity, while also laying the groundwork for further evaluation such as differential testing. Ultimately, our 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 paper presents a first step in that direction by laying the programming languages foundation necessary to approach the problem in a rigorous and principled way.

The rest of the paper is organized as follows: Section 2 presents a brief overview of 3D printing, which is one of the most popular desktop class manufacturing techniques and the primary motivation behind our synthesis tool. Section 3 describes a model of CAD programming as a purely functional programming language equipped with a denotation to 3D solids. It also presents a formalism of triangular polygon meshes along with a denotation to 3D solids. Section 4 presents a meaning-preserving compilation algorithm from CAD to STL and a proof sketch for compiler correctness. Section 5 presents our synthesis algorithm for reverse compiling STL to CAD, along with a definition and proof of correctness which introduces the technique of "geometric oracles" to reason about the algorithm. Section 6 presents a set of case studies demonstrating the feasibility of these ideas by synthesizing editable CAD from unstructured STL objects selected from Thingiverse. Section 8 describes key areas for future work. Section 9 concludes.

[1]Our tool is available here: http://incarnate.uwplse.org/

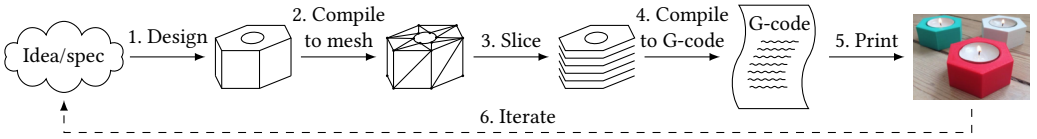


Fig. 1. 3D printing workflow depicting the 6 steps described in Section 2.1.

2 3D PRINTING BACKGROUND AND MOTIVATION

Among various desktop-class manufacturing devices, 3D printers are the most widely adopted due partly to the fact that they are relatively safer to use compared to desktop versions of other devices such as laser cutters and CNC mills. Figure 1 shows a 3D printing workflow, which typically comprises six steps: (1) design, (2) compile to mesh, (3) slice, (4) compile to G-code, (5) print, and (6) iterate.

2.1 3D printing as Compilation

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 2018], Rhino [Rhinoceros 2018], or SketchUp [SketchUp 2018]. This compilation process is complex and, similar to classical compilers, typically proceeds through a sequence of intermediate languages that we now describe.

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).

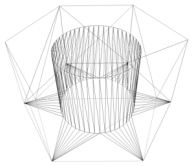
In most desktop-class 3D printers, a spool feeds filament (typically a plastic) into an *extruder*, which heats and melts the filament before extruding it through a nozzle onto a *print bed*. It is this extruder that is controlled by the G-code, via low-level commands that actuate stepper motors to move 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 (see Section 2.2).

2.2 Synthesis Example

Consider the model of a hexagonal candle holder from Thingiverse [Thingiverse 2018a] shown in Figure 2a. Like most models shared in online repositories, it is shared as a mesh. (Figure 2b shows the rendering of the mesh.) Figure 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) 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.

Fig. 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.

```
Diff (
  Scale (4.0, 4.0, 2.0) (
    Cylinder(6)
  )
  Scale (2.0, 2.0, 3.0) (
    Translate (0, 0, 0.3) (
      Cylinder(50)
    )
  )
)
```

(a) λCAD program for candle holder.

```
Diff (
  Scale (4.0, 4.0, 2.0) (
    Cylinder(6)
  )
  Scale (2.5, 2.5, 3.5) (
    Translate (0, 0, 0.3) (
      Cylinder(50)
    )
  )
)
```

(b) Edited λCAD program in blue.

Fig. 3. λCAD program for the hexagonal candle holder in Figure 2, and example of a modification that changes the dimension of the hole (in blue). The hexagonal outer part is represented by `Cylinder(6)` (a cylindrical prism with 6 sides), and the hole is represented by `Cylinder(50) (approximation of a cylinder using 50 sides)`. As shown, in λCAD this edit is done by changing the scaling factor for the cylindrical hole.

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 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 (`Cylinder(6)`) and subtract a (high-degree approximation of a) cylindrical hole (`Cylinder(50)`) from its center. With access to this program, editing it is straightforward. Figure 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

$$
\begin{array}{rcl}
C & ::= & \mathit{Mesh}\ M \\
& | & \mathit{Empty} \\
& | & \mathit{Cube} \\
& | & \mathit{Cylinder}\ \mathbb{N} \\
& | & \ldots \\
& | & \mathit{Affine}\ \mathbb{R}^{3x3}\ \mathbb{R}^3\ C \\
& | & \mathit{Binop}\ \mathrm{op}\ C\ C
\end{array}
\qquad
\begin{array}{rcl}
M & ::= & (\mathbb{R}^3, \mathbb{R}^3, \mathbb{R}^3)^* \\
\\
\mathit{op} & ::= & \mathit{Union} \\
& | & \mathit{Inter} \\
& | & \mathit{Diff}
\end{array}
$$

$$
\begin{array}{rcl}
[\![\, \mathit{Mesh}\ m \,]\!] & = & [\![\, m \,]\!] \\
[\![\, \mathit{Empty} \,]\!] & = & \{\} \\
[\![\, \mathit{Cube} \,]\!] & = & (0,1)^3 \\
[\![\, \mathit{Affine}\ p\ q\ c \,]\!] & = & \{pv + q \mid v \in [\![\, c \,]\!]\} \\
[\![\, \mathit{Binop}\ o\ c_1\ c_2 \,]\!] & = & [\![\, c_1 \,]\!]\ [\![\, o \,]\!]\ [\![\, c_2 \,]\!] \\
[\![\, \mathit{Union} \,]\!] = \cup \quad [\![\, \mathit{Inter} \,]\!] = \cap \quad [\![\, \mathit{Diff} \,]\!] = \setminus
\end{array}
$$

Fig. 4. CAD syntax and semantics. CAD programs denote to regular open sets in $\mathbb{R}^3$. Affine transformations are given by an *invertible* 3×3 matrix and translation vector. Binary operators denote to set operations. Mesh denotation is detailed below.

describe the first synthesis algorithm (Section 5) that automatically finds a CAD program from a surface mesh. This approach gives the users a high level CAD program to get started with, prevents them from having to make tedious mesh modifications while still allowing them to download mesh models from the internet.

3 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.

3.1 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 4 shows the syntax of the geometric core of λCAD. It supports (1) 3D primitives such as *Cube* and *Cylinder*, (2) affine transformations such as translation (*Translate*), rotation about X, Y, and Z axes (*RotateX*, *RotateY*, *RotateZ*), uniform and non-uniform scaling (*Scale*), and, (3) set-theoretic operations: *Union*, *Difference*, and *Intersection*. The primitives represent 3D shapes with unit measures. For example, *Cube* has all sides of unit length and the bottom left corner at the origin, $(0, 0, 0)$. *Cylinder*(n) is an n-sided prism with unit radius and height whose base is centered at the origin. Note that, as presented, all primitive objects are piecewise linear, thus requiring curves to be approximated. Truly curved primitives are interesting and possible, but complicate the semantics, compilation, and synthesis approaches discussed in this paper. 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. In this paper, we represent curves using approximations—for example, to represent a cylinder we set $n = 50$ in *Cylinder*(n). All the affine transformations are represented using an invertible 3×3 matrix and a 3D vector in the core CAD syntax in Figure 4. λCAD also supports user-provided raw meshes using the *Mesh* construct.

Figure 4 describes a denotational semantics for CAD that maps each object to the set of 3D points inside it. The primitive *Empty* maps to the empty set, while *Cube* maps to the set of all points whose

$$
\begin{array}{rcl}
x, y, z & \in & \mathbb{R} \\
\\
pt & \in & \text{point} \\
 & ::= & (x, y, z) \\
\\
f & \in & \mathit{Face} \\
 & ::= & (pt, pt, pt) \\
\\
m & \in & \mathit{Mesh} \\
 & ::= & f^*
\end{array}
$$

(a) Syntax of Mesh.

$$
\begin{array}{rcl}
r & \in & \mathbb{R} \\
d & \in & \mathit{Direction} \\
d & ::= & (r, r, r) \\
h & \in & \mathit{HalfLine} \\
h & ::= & (pt, d) \\
\text{Midpoint} & : & \mathit{Face} \to \text{point} \\
\text{Norm} & : & \mathit{Mesh} \times \mathit{Face} \to \{L, R\} \\
\text{On} & : & \mathit{Mesh} \times \mathit{Face} \to \text{bool}
\end{array}
$$

(b) Mesh functions used in the compiler (Figure 9).

Fig. 5. Syntax and auxiliary definitions for mesh. In Figure 5b, Midpoint is the centroid of a face. A point is On a face if it is coplanar with the face, and is in the interior of the face or on one of its edges.

x, y, and z values lie in $(0, 1)$. 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. Intersection and difference are similar.

3.2 Surface Mesh

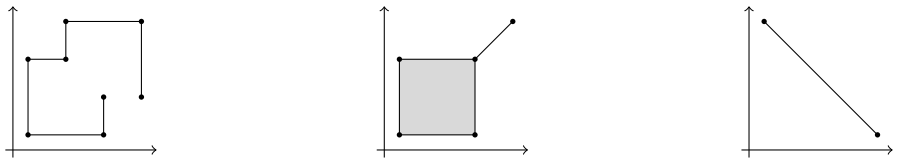
A surface polygon mesh is a geometric representation of the surface of an object in 3D space using *vertices*, *edges*, and *faces*. The faces of a mesh are typically convex polygons. In this paper, we formalize triangular meshes as shown in Figure 5a. A mesh is a list of faces, each of which is a triangle represented by its three vertices. 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 3.2.2 describes how we use *normals* to determine which side of a triangular face is inside/outside a 3D object.

3.2.1 Valid Mesh. In order for a 3D CAD model to be printable, the mesh should be *valid*. 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
- no edges that occur in an odd number of faces
- not open, i.e. should not have holes (or missing faces). This happens if an edge is on an odd number of faces.

Figure 6 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 6 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.

3.2.2 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. We use L (left) and R (right) to indicate the two possible directions for normals (Figure 5b), depending on whether the left-hand or right-hand rule should be used on the given face. Typical industrial



(a) 2D model with missing face. (b) 2D model with extra face. (c) Flat 2D model with zero area

Fig. 6. 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.

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 whether a point is inside or outside 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 5b. 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 7). 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, *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, 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.

This 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 7 defines this semantics based on face and 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. `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

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

Fig. 7. Semantics of Mesh using intersection of faces with halflines (rays).

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`.

4 3D PRINTING AS COMPILATION

This section presents a meaning-preserving compiler that generates a triangular mesh from 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.

Figure 9 defines the compiler as a recursive function on the syntax of the CAD program. 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 9. Since we use pre-defined meshes to approximate curves in this paper, 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.

4.1 Compiling CSG Operations and Mesh Splitting

To translate the set-theoretic binary operations, the compiler uses corresponding functions on meshes, $\mathit{mBop}(\mathit{Union})(m_1, m_2)$, $\mathit{mBop}(\mathit{Difference})(m_1, m_2)$, and $\mathit{mBop}(\mathit{Intersection})(m_1, m_2)$, shown in Figure 9. These operations are 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. Care is required to preserve the mesh invariants defined in Section 3.2.1.

The set-theoretic mesh operations first split the input meshes, defined by the relation *split* (used in Figure 9) that takes two potentially intersecting meshes and returns two equivalent meshes that intersection only at vertices and edges. The operation *split* preserves the semantics of the meshes, as proved in Theorem 4. The functions $\mathit{mBop}(\mathit{Union})(m_1, m_2)$, $\mathit{mBop}(\mathit{Difference})(m_1, m_2)$, and $\mathit{mBop}(\mathit{Intersection})(m_1, m_2)$ then 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 5b), if the face has the same *normal*, then it is kept, otherwise it is discarded. This is illustrated (in 2D) in Figure 8. $\mathit{mBop}(\mathit{Difference})(m_1', m_2')$ 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. $\mathit{mBop}(\mathit{Intersection})(m_1', m_2')$ keeps faces from m_1' that are inside m_2', faces from

(a) Keeping a common edge during mesh union. (b) Removing a common edge during mesh union.

Fig. 8. Examples (in 2D) demonstrating when common faces are retained or removed by *mBop*(*Union*).

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.

Definition 3. If f_i is a face on m_1 that overlaps with some face on m_2, *split* splits f_i in to $f_{i1}, f_{i2}, ..., f_{in}$ such that

$$\bigcup_{j=1}^{n} f_{ij} = f_i \quad \text{and} \quad \bigcap_{j=1}^{n} f_{ij} = \emptyset.$$

Theorem 4 (Mesh splitting correctness)**.** Given two valid meshes, m_1 and m_2, *split* (m_1, m_2, m_1', m_2') is a relation 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'. By definition, split ensures that the intersection of split faces are disjoint, so 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, 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'. □

4.2 Compiler Correctness

Our definition of compiler correctness is based on the denotational semantics we described in Section 3. Specifically, we prove that the compiler returns a 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.

Theorem 5 (Compiler correctness)**.** For all CAD expressions e, $[\![\mathit{compile}(e)]\!] = [\![e]\!]$.

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

Case *Empty*:

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

Case *Cube*:

$$\begin{aligned} [\![\mathit{compile}(\mathit{Cube})]\!] &= [\![m_{cube}]\!] && \text{By definition of } \mathit{compile}(). \\ &= [\![\mathit{Cube}]\!] && \text{By Lemma 6.} \end{aligned}$$

$$compile(Mesh\ m) = m, compile(Empty) = [], compile(Cube) = m_{cube}$$

$$\begin{array}{rcl}
C_2 & = & [((0,0),(1,0),(1,1)),((0,0),(0,1),(1,1))] \\
m_{cube} & = & [f(j, C_2) \mid f \in [i_x, i_y, i_z], j \in [0,1]] \\
i_x, i_y, i_z & : & \mathbb{R}^2 \to \mathbb{R}^3 \\
i_x(x_0,(a,b)) & = & (x_0, a, b) \\
i_y(y_0,(a,b)) & = & (a, y_0, b) \\
i_z(z_0,(a,b)) & = & (a, b, z_0) \\
compile(Affine\ p\ q\ c) & = & \mathsf{map}_{\mathsf{vertex}}\ (\lambda v.\ pv + q)\ (compile(c)) \\
compile(Binop\ o\ c_1\ c_2) & = & mBop(o)(compile(c_1),\ compile(c_2)) \\
mBop(Union)(m_1, m_2) & = & \mathsf{let}\ m_1', m_2'\ \text{s.t.,}\ split\ (m_1, m_2, m_1', m_2')\ \mathsf{in} \\
 & & [f \in m_1' \mid \nexists d.\ \mathsf{InsideVia}(m_2', \mathsf{Midpoint}(f), d)]\ ++ \\
 & & [f \in m_2' \mid \nexists d.\ \mathsf{InsideVia}(m_1', \mathsf{Midpoint}(f), d)]\ ++ \\
 & & [f \in m_1' \mid \mathsf{On}(m_2', f) \wedge \mathsf{Norm}(m_1', f) = \mathsf{Norm}(m_2', f)] \\
mBop(Difference)(m_1, m_2) & = & \mathsf{let}\ m_1', m_2'\ \text{s.t.,}\ split\ (m_1, m_2, m_1', m_2')\ \mathsf{in} \\
 & & [f \in m_1' \mid \nexists d.\ \mathsf{InsideVia}(m_2', \mathsf{Midpoint}(f), d)]\ ++ \\
 & & [f \in m_2' \mid \exists d.\ \mathsf{InsideVia}(m_1', \mathsf{Midpoint}(f), d)]\ ++ \\
 & & [f \in m_1' \mid \mathsf{On}(m_2', f) \wedge \mathsf{Norm}(m_1', f) \neq \mathsf{Norm}(m_2', f)] \\
mBop(Intersection)(m_1, m_2) & = & \mathsf{let}\ m_1', m_2'\ \text{s.t.,}\ split\ (m_1, m_2, m_1', m_2')\ \mathsf{in} \\
 & & [f \in m_1' \mid \exists d.\ \mathsf{InsideVia}(m_2', \mathsf{Midpoint}(f), d)]\ ++ \\
 & & [f \in m_2' \mid \exists d.\ \mathsf{InsideVia}(m_1', \mathsf{Midpoint}(f), d)]\ ++ \\
 & & [f \in m_1' \mid \mathsf{On}(m_2', f) \wedge \mathsf{Norm}(m_1', f) = \mathsf{Norm}(m_2', f)]
\end{array}$$

Fig. 9. Representative cases of CAD-to-Mesh compiler. Midpoint, On and Norm are as defined in Figure 5b.

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

$$\begin{array}{rcll}
[\![\ compile(Affine\ p\ q\ e')\ ]\!] & = & [\![\ \mathsf{map}_{\mathsf{vertex}}\ (\lambda v.\ pv + q)\ (m')\ ]\!] & \text{By definition of } compile(). \\
 & = & \{pv + q \mid v \in [\![\ m'\ ]\!]\} & \text{By Lemma 7.} \\
 & = & \{pv + q \mid v \in [\![\ e'\ ]\!]\} & \text{By induction hypothesis.} \\
 & = & [\![\ Affine\ p\ q\ e'\ ]\!] & \text{By definition of CAD } [\![\ ]\!].
\end{array}$$

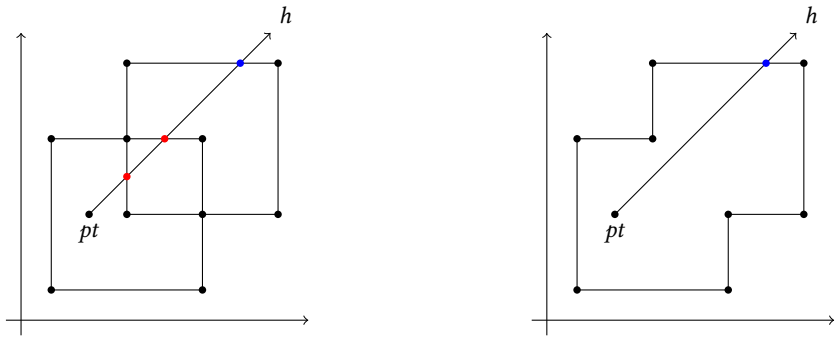
Case *Union* e_1 e_2: Let m_1, m_2 represent $compile(e_1)$ and $compile(e_2)$ respectively.

$$\begin{array}{rcll}
[\![\ compile(Binop\ Union\ e_1\ e_2)\ ]\!] & = & [\![\ mBop(Union)(m_1, m_2)\ ]\!] & \text{By definition of } compile(). \\
 & = & [\![\ m_1\ ]\!] \cup [\![\ m_2\ ]\!] & \text{By Lemma 8.} \\
 & = & [\![\ e_1\ ]\!] \cup [\![\ e_2\ ]\!] & \text{By induction hypothesis.} \\
 & = & [\![\ Union\ e_1\ e_2\ ]\!] & \text{By definition of CAD } [\![\ ]\!].
\end{array}$$

□

Lemma 6. $[\![\ m_{cube}\ ]\!] = [\![\ Cube\ ]\!]$

Proof. (⊆) 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



(a) Before (b) After

Fig. 10. Examples demonstrating (in 2D) several cases in the proof of Lemma 8.

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.

(⊇) 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 7. 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 8. 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 10.

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.

□

4.3 Implementation and Challenges

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.

4.3.1 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 4.1. On the other hand, the technique for finding the intersection of a face and a halfline in Section 3.2.2 is 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 11 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 3.2, a *valid* 1D mesh should not have repeating faces or odd number of faces.

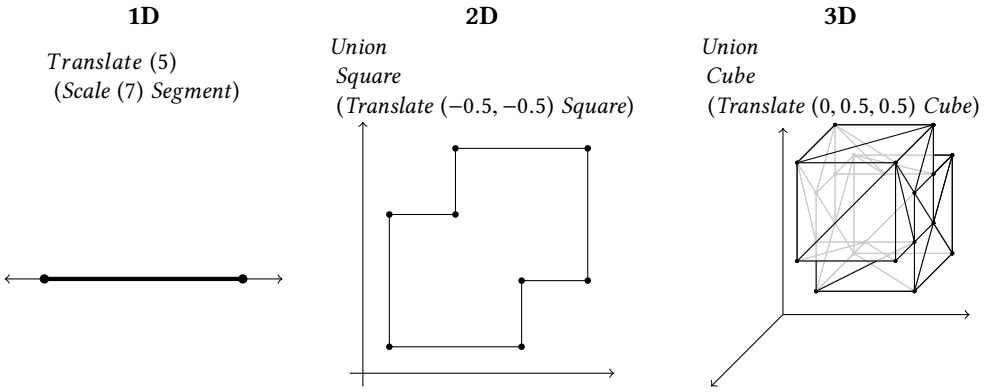


Fig. 11. Examples of 1D, 2D, 3D CADs and meshes. In order to keep the figures simple, the axes are not shown to intersect at the origin.

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 11 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, `rotateX`, `rotateY`, `rotateZ`. Translation and scaling are obviously possible, but using 3D vectors. Figure 11 shows a 3D CAD program and the corresponding triangular mesh.

4.3.2 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 2018], 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. We have implemented this signature using floating points, arbitrary precision number systems such as MPFR [Fousse et al. 2007; Zimmermann 2010], and exact arithmetic (see Section 8.1).

5 SYNTHESIZING EDITABLE CAD BY REVERSE COMPILATION

To demonstrate how the foundations established in previous sections can help develop better tools for desktop-manufacturing users, we describe a novel algorithm for "reverse compiling" meshes to CAD programs that recaptures the high-level structure of a design. In this section we show how our CAD and mesh formalizations suggest a natural search strategy for synthesis. In Section 6 we detail case studies that demonstrate the promise of this approach: once meshes are reverse

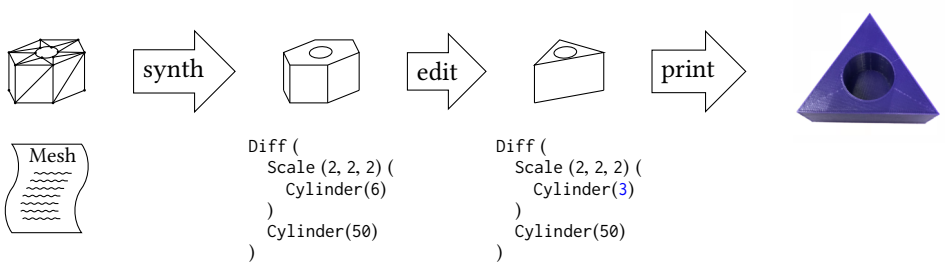


Fig. 12. Synthesis workflow: starting with a mesh for a 3D model, our synthesis tool can reverse engineer a CAD program, which can be easily edited to get a different 3D model.

$$E ::= [\cdot] \mid \mathit{Affine}\ \mathbb{R}^{3\times3}\ \mathbb{R}^3\ E \mid \mathit{Binop}\ \mathrm{op}\ E\ C \mid \mathit{Binop}\ \mathrm{op}\ (\mathit{Mesh}\ \mathrm{m})\ E$$

$$S ::= A \mid \mathit{Binop}\ (\mathit{Union} \mid \mathit{Diff})\ S\ S \qquad A ::= P \mid \mathit{Affine}\ \mathbb{R}^{3\times3}\ \mathbb{R}^3\ A \qquad P ::= \mathit{Cube} \mid \mathit{Cylinder}\ \mathbb{N} \mid \ldots$$

$$\frac{c \to_p c'}{E[c] \to_c E[c']} \qquad \frac{}{\mathit{Cube} \to_p \mathit{Mesh}\ m_{cube}} \qquad \frac{m \to_\Omega c}{E[\mathit{Mesh}\ m] \to_s E[c]} \qquad \frac{p \in \Omega_{\mathrm{prim}}(m)}{\mathit{Mesh}\ m \to_\Omega p}$$

$$\frac{\mathrm{map}_{\mathrm{vertex}}\ (\lambda v.\ pv + q)\ m = m'}{\mathit{Affine}\ p\ q\ (\mathit{Mesh}\ m) \to_p \mathit{Mesh}\ m'} \qquad \frac{(m_1, m_2) \in \Omega_{\mathrm{add}}(m)}{\mathit{Mesh}\ m \to_\Omega \mathit{Binop}\ \mathit{Union}\ (\mathit{Mesh}\ m_1)\ (\mathit{Mesh}\ m_2)}$$

$$\frac{\mathit{mBop}(o)(m_1, m_2) = m'}{\mathit{Binop}\ o\ (\mathit{Mesh}\ m_1)\ (\mathit{Mesh}\ m_2) \to_p \mathit{Mesh}\ m'} \qquad \frac{(m_1, m_2) \in \Omega_{\mathrm{sub}}(m)}{\mathit{Mesh}\ m \to_\Omega \mathit{Binop}\ \mathit{Diff}\ (\mathit{Mesh}\ m_1)\ (\mathit{Mesh}\ m_2)}$$

Fig. 13. Representative cases of small step CAD compilation ($\to_c$, left) and synthesis ($\to_s$, right) with evaluation contexts and synthesis target language (E, S).

compiled to CAD, many edits which would be tedious or prohibitively difficult at the mesh level become trivial. For example, Figure 12 shows an example of creating a triangle candle holder by starting with a hexagonal one in mesh form, synthesizing CAD, then tweaking the number of sides in $\mathit{Cylinder}(n)$ before printing the desired object.

To develop our 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. Throughout the section we note key design insights specific to the CAD domain that focus the heuristics and shrink the search space to speed up synthesis.

5.1 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 14a. The resulting mesh only contains fragments of the cylinder primitives the programmer originally specified, yet intuitively we expect synthesis to "figure it out".

To both make this goal precise and support reasoning by induction on the synthesis search space, we first rephrase our CAD compiler as a small step relation $\rightarrow_c$ in Figure 13 which 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 13. 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:

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

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

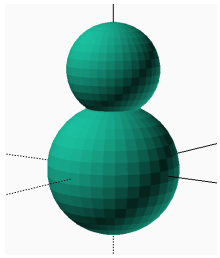
Synthesis is inherently under-constrained since there is never a unique CAD program that compiles to a given mesh, e.g., for all c, $[\![c]\!] = [\![\mathit{Binop\ Union}\ c\ c]\!]$. Furthermore, for any mesh m, there is a trivial "complete" synthesis strategy: simply map each face of m to the base of 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 13 where only union and difference CSG operations are allowed and are above all affine transformations which in turn are above all primitives. 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; we have found that a simple post-pass to factor out repeated affine operations can often address this issue. Furthermore, once a mesh has been synthesized up to the CAD level, traditional syntax-based synthesis techniques [Bornholt et al. 2016; Phothilimthana et al. 2016; Solar-Lezama 2008] could be applied to further improve editability and optimize other constraints.

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 $\mathit{synth}_\Omega(\mathit{compile}(\mathit{synth}_\Omega(m))) = \mathit{synth}_\Omega(m)$.

(a) Information lost in compilation.



(b) Composition depends on context.

Fig. 14. Figure 14a 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 14b shows how evaluation context can be used to synthesize the union of two spheres.

$$
\begin{aligned}
\mathit{synth}_\Omega(m) &= \mathit{search}_\Omega([\mathit{Mesh}\ m],\ [],\ \mathit{fuel}) \\
\mathit{search}_\Omega(cs,\ \mathit{fs},\ 0) &= \max_{\leq_{\text{edit}}}\ (\mathit{fs}\ +\!\!+\ cs) \\
\mathit{search}_\Omega([],\ \mathit{fs},\ \mathit{fuel}) &= \max_{\leq_{\text{edit}}}\ \mathit{fs} \\
\mathit{search}_\Omega(c :: cs,\ \mathit{fs},\ \mathit{fuel}) &= \\
&\quad \text{let } E[\mathit{Mesh}\ m] = \mathit{focus}(c) \text{ in} \\
&\quad \text{let } \mathit{ps} = \text{map } (\lambda c.\ E[c])\ \ \Omega_{\text{prim}}(m) \text{ in} \\
&\quad \text{let } \mathit{as} = \text{map } (\lambda(m_1, m_2).\ E[\mathit{Binop\ Union}\ (\mathit{Mesh}\ m_1)\ (\mathit{Mesh}\ m_2)])\ \ \Omega_{\text{add}}(m) \text{ in} \\
&\quad \text{let } \mathit{ss} = \text{map } (\lambda(m_1, m_2).\ E[\mathit{Binop\ Diff}\ (\mathit{Mesh}\ m_1)\ (\mathit{Mesh}\ m_2)])\ \ \Omega_{\text{sub}}(m) \text{ in} \\
&\quad \text{let } (\mathit{fs}', \mathit{cs}') = \text{partition } (\lambda c.\ c \in \mathrm{S})\ \ (\mathit{ps}\ +\!\!+\ \mathit{as}\ +\!\!+\ \mathit{ss}) \text{ in} \\
&\quad \text{let } \mathit{cs}'' = \text{fold } \mathit{schedule}\ \mathit{cs}\ \mathit{cs}' \text{ in} \\
&\quad \mathit{search}_\Omega(\mathit{cs}'',\ c :: \mathit{fs}'\ +\!\!+\ \mathit{fs},\ \mathit{fuel} - 1)
\end{aligned}
$$

Fig. 15. Core synthesis algorithm.

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

5.2 Algorithm

Algorithm 15 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 $\leq_{\text{edit}}$.

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 $\leq_{\text{edit}}$ 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 13, 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 mesh using the inferred canonicalization parameters and returns the result. We describe canonicalization in more detail in Section 5.4.1.

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. 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.

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 more 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.

5.2.1 Context and Sharing. The context of a mesh plays a critical role in synthesis. For example, consider the case of a union of two overlapping spheres[2] in Figure 14b. Ω_{add} will use convex splitting to break union into two *truncated spheres*. No affine transformation of a primitive can match a truncated sphere. 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

[2]As mentioned in Section 3.1, this paper approximates curves. To represent the sphere, we use a predefined mesh.

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, as we explained in Section 4.

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.

5.3 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.

□

5.4 Implementation and Challenges

We implemented our synthesis tool, ReIncarnate to work with the compiler tools we built in Section 4.3. Due to our full functorial design strategy (Section 4.3.2), we were able to implement synthesis as an extension to the existing system by making `Synthesis` a functor over `NumSys`, `Geometry`, `Mesh` and `CAD`.

5.4.1 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, Ω_{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 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:

$$m_r = \texttt{rotateX}(-r_x)\ (\texttt{rotateY}(-r_y)\ (\texttt{rotateZ}(-r_z)\ m))$$

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:

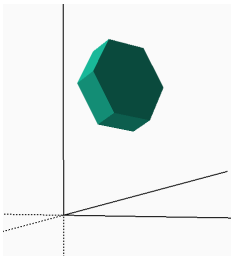
$$p' = \texttt{rotateZ}(r_z)(\texttt{rotateY}(r_y)(\texttt{rotateX}(r_x)(\texttt{scale}(d_x, d_y, d_z)\ p)))$$

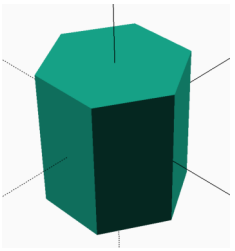
The last step is to translate the scaled and rotate primitive, p' to the right coordinates. The distance to be translated is the distance between the center of $compile(p')$ and the center of the original mesh m: $(c_x^{p'}, c_y^{p'}, c_z^{p'}) - (c_x^m, c_y^m, c_z^m)$.

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

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.

5.4.2 A Concrete Instance of the Synthesis Algorithm. Following is a concrete example of how the ReIncarnate algorithm in Figure 15 works. Consider the mesh m of the model in Figure 16 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 15, 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 16 (second figure) shows the canonicalized mesh. Since this already matches with an affine transformed primitive (`Cylinder(6)`), *ps* will be a list containing the corresponding λCAD program. This program is shown in Figure 16. 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 16, and cs'





```
Translate(2.4, 0.8, 5.2) (
  RotateZ(80.5) (
    RotateY(−40.5) (
      RotateX(−124.7) (
        Scale(2.0, 1.7, 1.0) (
          Cylinder (6) )))))
```

Fig. 16. Canonicalization and λCAD synthesis of arbitrary object in 3D space.

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 15, it will now apply $\max_{\leq_{\text{edit}}}$ to *fs*. Since *fs* contains only one λCAD program (shown in Figure 16), this will be the output of synthesis.

5.4.3 Simplify. We implemented a recursive *simplification* algorithm, that, given a CAD AST, traverses it to remove redundant nodes. Synthesis can return CAD programs that have redundant nodes in the AST. For example, *Binop Union c c* is equivalent to *c* for any CAD program *c*. To remove such redundant nodes, the CAD program is passed on to *simplify* as the final step. For primitives shapes, *simplify* simply returns the same node. For nodes that are affine transformations, *simplify* compiles the child node and if the mesh thus obtained is the same as the mesh *with* the affine transformation, then *simplify* removes the affine node. For binary operations, *simplify* compiles the left child first. If the mesh obtained is the same as the mesh *with* the binary operation, then *simplify* returns the left child. If not, it tries the same with the right child. If that fails, then *simplify* will return the original AST with the binary operation node. Since *simplify* compiles the child nodes in every recursive step, it can slow down the performance of the synthesis algorithm for very large ASTs. That is why ReIncarnate only uses *simplify* on the final λCAD program and not at the intermediate stages of synthesis.

6 CAD SYNTHESIS CASE STUDIES

We demonstrate three case studies on which we ran ReIncarnate. Two of the case studies are downloaded from Thingiverse and one is our own design (Table 1). 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.
- *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

Table 1. Summary of case studies.

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

```
Diff (
  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
        )
      )
    )
  )
)
```

Fig. 17. λCAD program for **I** synthesized by ReIncarnate. Rendered λCAD programs for ICFP and CFP.

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.

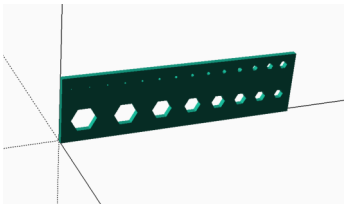
6.1 ICFP

This model (shown in Figure 17) 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 17 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 17. Figure 19 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,

```
Diff (
  Translate(0, 0, 0.5) (
    Scale(2, 1.732, 1) (
      Translate(-0, -0, -0.5) (
        Scale(0.5, 0.57, 1.0) (
          Cylinder(6)
        )
      )
    )
  )
  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(50)
        )
      )
    )
  )
)
```

(a) Synthesized λCAD for the candle holder.



(b) Rendered hex holder.

Fig. 18. Figure 18a shows the λCAD program that ReIncarnate synthesized for the candle holder [Thingiverse 2018a]. Figure 18b shows the hex wrench holder from Thingiverse [Thingiverse 2018c].

this seemingly simple task can get confusing and tedious because the user has to drag the letters around to make the spacing uniform.

6.2 Candle Holder

Our next example is the candle holder (shown in Figure 2 in Section 2) 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 18a.

- 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 [Blender 2018; Meshmixer 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 (Figure 3).
- 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 *RotateX*, *RotateY*.
- 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 a single character change to the argument n of `Cylinder(`n`)`) but as difficult as manually editing the triangular faces at the mesh level.

6.3 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 19 shows a part of the λCAD program.

- One can use *RotateZ(θ)* as shown in Figure 19 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.

Synthesizing λCAD for this model opens up directions for future work on CAD synthesis. Figure 18b shows that 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. ReIncarnate can be extended using classic program synthesis techniques [Bornholt et al. 2016; Phothilimthana et al. 2016] to detect such repetitions and optimize the generated λCAD programs.

7 RELATED WORK

To the best of our knowledge, this is the first work that (1) relates the semantics of CAD and surface meshes using programming-languages techniques and (2) uses program synthesis for reverse engineering CAD from surface meshes. 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.

7.1 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 [Rhinoceros 2018; SketchUp 2018; Solidworks 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 2018], which is a popular programmatic CAD tool in the 3D design sharing community [Thingiverse 2018b]. OpenSCAD builds on the CGAL [CGAL 2018] computational geometry library. There are other programmatic CAD languages such as ImplicitCAD [ImplicitCAD 2018] 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

Synthesized λCAD for hex holder:

```
Diff (
  Translate(65.0, 15.0, 2.5) (
    Scale(130.0, 30.0, 5.0) (
      Translate(-0.5, -0.5, -0.5) (
        Cube
      )
    )
  )
  Translate(5.0, 23.0, 2.5) (
    Scale(2.0, 1.7, 0.5) (
      Cylinder(6)
    )
  )
  ...)
```

Adding rotation about Z on line 9:

```
Diff (
  Translate(65.0, 15.0, 2.5) (
    Scale(130.0, 30.0, 5.0) (
      Translate(-0.5, -0.5, -0.5) (
        Cube
      )
    )
  )
  RotateZ(15.0) (
    Translate(5.0, 23.0, 2.5) (
      Scale(2.0, 1.7, 5.0) (
        Cylinder(6)
      )
    )
  )
  ...)
```

λCAD synthesized for ICFP after removing I:

```
Union (
  Union (
    (* C *)
    Diff (
      Translate(2.0, 0.5, 0.5) (
        Translate(-0.5, -0.5, -0.5) (
          Cube ))
      Translate(2.175, 0.5, 0.5) (
        Scale(0.65, 0.5, 1.0) (
          Translate(-0.5, -0.5, -0.5) (
            Cube ))))
    (* F *)
    Diff (
      Translate(3.5, 0.5, 0.5) (
        Translate(-0.5, -0.5, -0.5) (
          Cube ))
      Diff (
        Translate(3.6, 0.4, 0.5) (
          Scale(0.8, 0.8, 1.0) (
            Translate(-0.5, -0.5, -0.5) (
              Cube )))
        Translate(3.4, 0.5, 0.5) (
          Scale(0.4, 0.2, 1.0) (
            Translate(-0.5, -0.5, -0.5) (
              Cube ))))))
  (* P *)
  Diff (
    Translate(5.0, 0.5, 0.5) (
      Translate(-0.5, -0.5, -0.5) (
        Cube ))
    Union (
      Diff (
        Translate(5.15, 0.15, 0.5) (
          Scale(0.7, 0.3, 1.0) (
            Translate(-0.5, -0.5, -0.5) (
              Cube )))
        Empty )
      Diff (
        Translate(5.05, 0.65, 0.5) (
          Scale(0.5, 0.3, 1.0) (
            Translate(-0.5, -0.5, -0.5) (
              Cube )))
        Empty ))))
```

Fig. 19. 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**. This program is the simply the union of the three letters.

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. 2013] 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. 2015, 2016] 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].

7.2 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.

7.3 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 2018d] 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.

7.4 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 [Baldwin et al. 2017; Banovic et al. 2013; Guo et al. 2017; Hofmann et al. 2016b; Hofmann 2015]. 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].

7.5 Program Synthesis

Program synthesis is applied to a wide range of applications such as super optimizations for low power spatial architectures [Phothilimthana et al. 2014, 2016], 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 2008] 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.

Our synthesis algorithm can be viewed as a reverse-compilation process that starts with a mesh representation of a 3D model as a specification, and searches for a CAD representation of the same. The oracles described in Section 5.2 navigate the search in the right direction. Unlike both traditional deductive and inductive synthesis, neither meshes nor CAD programs take inputs or produce outputs.

8 FUTURE WORK

In this section we briefly survey 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.

8.1 Exact Arithmetic

One challenge to implementing the semantics described in Sections 3.1 and 3.2 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.

8.2 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. Conversely, our current synthesis procedure never uses hull. Hull is a powerful primitive, and it is unclear in what situations its use helps or hinders producing editable CAD objects.

8.3 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 8.1, 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.

8.4 Compiling down to G-code

Section 2 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.

9 CONCLUSIONS

We presented a functional-programming approach to designing and implementing computer-aided design tools. We formalized CAD and surface mesh and provided denotational semantics to both. We implemented a compiler for the semantics and sketched its correctness. We proposed a synthesis technique that uses reverse engineering and geometric oracles to provide CAD programs from surface meshes and showed that it works on real surface meshes downloaded from one of the most popular online repositories for 3D models (Thingiverse). 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.

ACKNOWLEDGMENTS

Thanks to Sarah Chasins, Martin Kellogg, Stuart Pernsteiner, Talia Ringer, Jared Roesch, Ben Sherman, Remy Wang, and Doug Woos for their feedback on early drafts of the paper. We thank John Toman for useful advice on the artifact. We thank Eva Darulova and Max Willsey for numerous stimulating technical conversations. We are grateful to the anonymous reviewers for providing valuable comments.

REFERENCES

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

Rajeev Alur, Loris D'Antoni, Sumit Gulwani, Dileep Kini, and Mahesh Viswanathan. 2013. Automated Grading of DFA Constructions. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence (IJCAI '13)*. AAAI Press, 1976–1982. http://dl.acm.org/citation.cfm?id=2540128.2540412

M. Artin. 2011. *Algebra*. Pearson Prentice Hall. https://books.google.com/books?id=S6GSAgAAQBAJ

Mark S. Baldwin, Gillian R. Hayes, Oliver L. Haimson, Jennifer Mankoff, and Scott E. Hudson. 2017. The Tangible Desktop: A Multimodal Approach to Nonvisual Computing. *ACM Trans. Access. Comput.* 10, 3, Article 9 (Aug. 2017), 28 pages. https://doi.org/10.1145/3075222

Nikola Banovic, Rachel L. Franz, Khai N. Truong, Jennifer Mankoff, and Anind K. Dey. 2013. 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)*. ACM, New York, NY, USA, Article 24, 8 pages. https://doi.org/10.1145/2513383.2513445

Blender. 2018. Blender. (2018). https://www.blender.org/.

James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. 2016. Optimizing Synthesis with Metasketches. *SIGPLAN Not.* 51, 1 (Jan. 2016), 775–788. https://doi.org/10.1145/2914770.2837666

CGAL. 2018. CGAL. (2018). https://www.cgal.org.

Xiang 'Anthony' Chen, Stelian Coros, Jennifer Mankoff, and Scott E. Hudson. 2015. 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*. 3:1. https://doi.org/10.1145/2787626.2787650

Xiang 'Anthony' Chen, Jeeeun Kim, Jennifer Mankoff, Tovi Grossman, Stelian Coros, and Scott E. Hudson. 2016. 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*. 29–39. https://doi.org/10.1145/2984511.2984512

M. de Berg. 1997. *Computational Geometry: Algorithms and Applications*. Springer. https://books.google.com/books?id=_vAxRFQcNA8C

P. Delfs, M. T̈ows, and H.-J. Schmid. 2016. Optimized build orientation of additive manufactured parts for improved surface quality and build time. *Additive Manufacturing* 12, Part B (2016), 314 – 320. https://doi.org/10.1016/j.addma.2016.06.003 Special Issue on Modeling & Simulation for Additive Manufacturing.

James Demmel and Yozo Hida. 2004. Fast and Accurate Floating Point Summation with Application to Computational Geometry. *Numerical Algorithms* 37, 1 (01 Dec 2004), 101–112. https://doi.org/10.1023/B:NUMA.0000049458.99541.38

Jérémie Dumas, An Lu, Sylvain Lefebvre, Jun Wu, and Christian Dick. 2015. By-example Synthesis of Structurally Sound Patterns. *ACM Trans. Graph.* 34, 4, Article 137 (July 2015), 12 pages. https://doi.org/10.1145/2766984

Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. 2007. MPFR: A Multiple-precision Binary Floating-point Library with Correct Rounding. *ACM Trans. Math. Softw.* 33, 2, Article 13 (June 2007). https://doi.org/10.1145/1236463.1236468

Salomé Galjaard, Sander Hofman, and Shibo Ren. 2015. *New Opportunities to Optimize Structural Designs in Metal by Using Additive Manufacturing*. Springer International Publishing, Cham, 79–93. https://doi.org/10.1007/978-3-319-11418-7_6

Geomagic Design X. 2018. Geomagic Design X. (2018). https://www.3dsystems.com/software/geomagic-design-x.

David Goldberg. 1991. What Every Computer Scientist Should Know About Floating-point Arithmetic. *Comput. Surveys* 23, 1 (March 1991), 5–48. http://doi.acm.org/10.1145/103162.103163

GrabCAD. 2018. GrabCAD. (2018). https://grabcad.com/.

T. Grimm. 2004. *User's Guide to Rapid Prototyping*. Society of Manufacturing Engineers.

Anhong Guo, Jeeeun Kim, Xiang 'Anthony' Chen, Tom Yeh, Scott E. Hudson, Jennifer Mankoff, and Jeffrey P. Bigham. 2017. Facade: Auto-generating Tactile Interfaces to Appliances. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17)*. ACM, New York, NY, USA, 5826–5838. https://doi.org/10.1145/3025453.3025845

Megan Hofmann, Julie Burke, Jon Pearlman, Goeran Fiedler, Andrea Hess, Jon Schull, Scott E. Hudson, and Jennifer Mankoff. 2016a. 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)*. ACM, New York, NY, USA, 251–256. https://doi.org/10.1145/2982142.2982181

Megan Hofmann, Jeffrey Harris, Scott E. Hudson, and Jennifer Mankoff. 2016b. 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)*. ACM, New York, NY, USA, 1769–1780. https://doi.org/10.1145/2858036.2858340

Megan Kelly Hofmann. 2015. 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)*. ACM, New York, NY, USA, 353–354. https://doi.org/10.1145/2700648.2811323

Kai Hormann and Alexander Agathos. 2001. The Point in Polygon Problem for Arbitrary Polygons. *Comput. Geom. Theory Appl.* 20, 3 (Nov. 2001), 131–144. https://doi.org/10.1016/S0925-7721(01)00012-8

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

ImplicitCAD. 2018. ImplicitCAD. (2018). http://www.implicitcad.org/.

Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-guided Component-based Program Synthesis. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10)*. ACM, New York, NY, USA, 215–224. https://doi.org/10.1145/1806799.1806833

Rajeev Joshi, Greg Nelson, and Keith Randall. 2002. Denali: A Goal-directed Superoptimizer. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI '02)*. ACM, New York, NY, USA, 304–314. https://doi.org/10.1145/512529.512566

Jeeeun Kim, Anhong Guo, Tom Yeh, Scott E. Hudson, and Jennifer Mankoff. 2017. 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*. 1067–1078. https://doi.org/10.1145/3064663.3064690

P. Kim. 2013. *Rigid Body Dynamics for Beginners: Euler Angles & Quaternions.* CreateSpace Independent Publishing Platform. https://books.google.com/books?id=bJEengEACAAJ

Venkat Krishnamurthy and Marc Levoy. 1996. Fitting Smooth Surfaces to Dense Polygon Meshes. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '96)*. ACM, New York, NY, USA, 313–324. https://doi.org/10.1145/237170.237270

James McCann, Lea Albaugh, Vidya Narayanan, April Grow, Wojciech Matusik, Jennifer Mankoff, and Jessica K. Hodgins. 2016. A compiler for 3D machine knitting. *ACM Trans. Graph.* 35, 4 (2016), 49:1–49:11. https://doi.org/10.1145/2897824.2925940

James McCrae, Nobuyuki Umetani, and Karan Singh. 2014. FlatFitFab: Interactive Modeling with Planar Sections. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology (UIST '14)*. ACM, New York, NY, USA, 13–22. https://doi.org/10.1145/2642918.2647388

Autodesk. Meshmixer. 2018. Autodesk. Meshmixer. (2018). http://www.meshmixer.com/.

Stefanie Mueller, Sangha Im, Serafima Gurevich, Alexander Teibrich, Lisa Pfisterer, François Guimbretière, and Patrick Baudisch. 2014a. WirePrint: 3D Printed Previews for Fast Prototyping. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology (UIST '14)*. ACM, New York, NY, USA, 273–280. https://doi.org/10.1145/2642918.2647359

Stefanie Mueller, Tobias Mohr, Kerstin Guenther, Johannes Frohnhofen, and Patrick Baudisch. 2014b. 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)*. ACM, New York, NY, USA, 3827–3834. https://doi.org/10.1145/2556288.2557005

Chandrakana Nandi, Anat Caspi, Dan Grossman, and Zachary Tatlock. 2017. Programming Language Tools and Techniques for 3D Printing. In *2nd Summit on Advances in Programming Languages (SNAPL 2017) (Leibniz International Proceedings in Informatics (LIPIcs))*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.), Vol. 71. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 10:1–10:12. https://doi.org/10.4230/LIPIcs.SNAPL.2017.10

OFF. 2018. OFF Files. (2018). http://www.geomview.org/docs/html/OFF.html.

OpenSCAD. 2018. OpenSCAD. (2018). http://www.openscad.org/.

Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. 2015. Automatically Improving Accuracy for Floating Point Expressions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 1–11. https://doi.org/10.1145/2737924.2737959

Phitchaya Mangpo Phothilimthana, Tikhon Jelvis, Rohin Shah, Nishant Totla, Sarah Chasins, and Rastislav Bodik. 2014. 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)*. ACM, New York, NY, USA, 396–407. https://doi.org/10.1145/2594291.2594339

Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. 2016. Scaling Up Superoptimization. *SIGPLAN Not.* 51, 4 (March 2016), 297–310. https://doi.org/10.1145/2954679.2872387

Powershape. 2018. Powershape. (2018). https://www.autodesk.com/products/powershape/overview.

Rhinoceros. 2018. Rhinoceros. (2018). https://www.rhino3d.com/.

Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic Superoptimization. *SIGPLAN Not.* 48, 4 (March 2013), 305–316. https://doi.org/10.1145/2499368.2451150

Adriana Schulz, Ariel Shamir, David I. W. Levin, Pitchaya Sitthi-amorn, and Wojciech Matusik. 2014. Design and Fabrication by Example. *ACM Trans. Graph.* 33, 4, Article 62 (July 2014), 11 pages. https://doi.org/10.1145/2601097.2601127

J.R. Shewchuk. 1997. Adaptive precision floating-point arithmetic and fast robust geometric predicates. 18 (10 1997), 305–363.

SketchUp. 2018. SketchUp. (2018). http://www.sketchup.com/.

P. Smid. 2003. *CNC Programming Handbook: A Comprehensive Guide to Practical CNC Programming*. Industrial Press. https://books.google.com/books?id=JNnQ8r5merMC

Armando Solar-Lezama. 2008. *Program Synthesis by Sketching*. Ph.D. Dissertation. University of California, Berkeley.

Solidworks. 2018. Solidworks. (2018). http://www.solidworks.com/.

SpaceClaim. 2018. SpaceClaim. (2018). http://www.spaceclaim.com/en/Solutions/ReverseEngineering.aspx.

Ondrej Stava, Juraj Vanek, Bedrich Benes, Nathan Carr, and Radomír Měch. 2012. Stress Relief: Improving Structural Strength of 3D Printable Objects. *ACM Trans. Graph.* 31, 4, Article 48 (July 2012), 11 pages. https://doi.org/10.1145/2185520.2185544

Ivan E. Sutherland. 1964. Sketch Pad a Man-machine Graphical Communication System. In *Proceedings of the SHARE Design Automation Workshop (DAC '64)*. ACM, New York, NY, USA, 6.329–6.346. https://doi.org/10.1145/800265.810742

Alexander Teibrich, Stefanie Mueller, François Guimbretière, Robert Kovacs, Stefan Neubert, and Patrick Baudisch. 2015. Patching Physical Objects. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology (UIST '15)*. ACM, New York, NY, USA, 83–91. https://doi.org/10.1145/2807442.2807467

Enabling The Future. 2018. Enabling The Future. (2018). http://enablingthefuture.org.

Thingiverse. 2018a. Hexagonal Candle Holder. (2018). https://www.thingiverse.com/thing:756968.

Thingiverse. 2018b. Thingiverse. (2018). http://www.thingiverse.com/.

Thingiverse. 2018c. Ultimate 22 Hex-Wrench Holder. (2018). https://www.thingiverse.com/thing:1752602.

Thingiverse. 2018d. Welcome To Customizer. (2018). https://www.thingiverse.com/customizer.

Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin, and Rajeev Alur. 2013. TRANSIT: Specifying Protocols with Concolic Snippets. *SIGPLAN Not.* 48, 6 (June 2013), 287–296. https://doi.org/10.1145/2499370.2462174

Nobuyuki Umetani and Ryan Schmidt. 2013. Cross-sectional Structural Analysis for 3D Printing Optimization. In *SIGGRAPH Asia 2013 Technical Briefs (SA '13)*. ACM, New York, NY, USA, Article 5, 4 pages. https://doi.org/10.1145/2542355.2542361

Kiril Vidimče, Szu-Po Wang, Jonathan Ragan-Kelley, and Wojciech Matusik. 2013. OpenFab: A Programmable Pipeline for Multi-material Fabrication. *ACM Trans. Graph.* 32, 4, Article 136 (July 2013), 12 pages. https://doi.org/10.1145/2461912.2461993

Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017. Synthesizing Highly Expressive SQL Queries from Input-output Examples. *SIGPLAN Not.* 52, 6 (June 2017), 452–466. https://doi.org/10.1145/3140587.3062365

Qingnan Zhou, Julian Panetta, and Denis Zorin. 2013. Worst-case Structural Analysis. *ACM Trans. Graph.* 32, 4, Article 137 (July 2013), 12 pages. https://doi.org/10.1145/2461912.2461967

Paul Zimmermann. 2010. Reliable Computing with GNU MPFR. In *Proceedings of the Third International Congress Conference on Mathematical Software (ICMS'10)*. Springer-Verlag, Berlin, Heidelberg, 42–45. http://dl.acm.org/citation.cfm?id=1888390.1888400