Sound and Robust Solid Modeling via Exact Real Arithmetic and Continuity

BENJAMIN SHERMAN, MIT, USA
JESSE MICHEL, MIT, USA
MICHAEL CARBIN, MIT, USA

Algorithms for solid modeling, i.e., Computer-Aided Design (CAD) and computer graphics, are often specified on real numbers and then implemented with finite-precision arithmetic, such as floating-point. The result is that these implementations do not soundly compute the results that are expected from their specifications.

We present a new library, StoneWorks, that provides sound and robust solid modeling primitives. We implement StoneWorks in MarshallB, a pure functional programming language for exact real arithmetic in which types denote topological spaces and functions denote continuous maps, ensuring that all programs are sound and robust. We developed MarshallB as an extension of the Marshall language.

We also define a new shape representation, *compact representation* (*K-rep*), that enables constructions such as Minkowski sum and analyses such as Hausdorff distance that are not possible with traditional representations. K-rep is a nondeterminism monad for describing all the points in a shape.

With our library, language, and representation together, we show that short StoneWorks programs can specify and execute sound and robust solid modeling algorithms and tasks.

CCS Concepts: • **Mathematics of computing** → *Arbitrary-precision arithmetic*; **Continuous functions**; *Point-set topology*; • **Theory of computation** → *Categorical semantics*; • **Applied computing** → *Computer-aided design.*

Additional Key Words and Phrases: Constructive Analysis, Synthetic Topology

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

99

1 INTRODUCTION

Computer-Aided Design (CAD) and computer graphics require *solid modeling*. Solid modeling is the design and implementation of computer representations of geometric shapes and the operations to construct and analyze them. For example, a CAD tool enables to user to construct a representation of mechanical system and verify that its constituent parts do not collide.

A solid modeling operation may return an incorrect answer because of the numerical error inherent in an implementation that uses floating-point arithmetic. Alternatively, an operation may not be robust because of an unstable algorithm for computing geometric properties, such as intersection. For instance, a CAD tool for detecting collisions may declare that a mechanical system is free of collisions, when in reality it is not. A ray-tracing algorithm may incorrectly cause thin features to disappear [Flórez et al. 2007]. A CAD tool may create a representation of a shape whose

Authors' addresses: Benjamin Sherman, MIT, USA, sherman@csail.mit.edu; Jesse Michel, MIT, USA, jmmichel@mit.edu; Michael Carbin, MIT, USA, mcarbin@csail.mit.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2019 Copyright held by the owner/author(s).
2475-1421/2019/8-ART99
https://doi.org/10.1145/3341703

topological and geometric data are inconsistent, meaning no real shape could have such properties. Inaccuracy in solid modeling "can sometimes lead to system crashes, which is frustrating to users, hinders productivity and automation efforts, and might result in costly expense" [Hu et al. 1996]. And "due to rounding errors many implementations of geometric algorithms crash, loop forever, or in the best case, simply compute wrong results for some of the inputs for which they are supposed to work" [Schirra 1998].

1.1 State of the Practice

Boundary representation (B-rep) is the dominant representation used by commercial CAD engines. It represents faces, edges, and vertices, and the topological relations between them, with explicit geometric representations. The rich information provided by boundary representation allows many operations to be performed, such as *extrusion*, *chamfer*, and *blending*. However, because of the richness of the representation, sound implementations of these operations are not computable. Moreover, best-effort implementations of these kernels are not robust.

For instance, consider the union of two squares whose edges are almost touching: if the edges are touching, then the faces should be merged into a single one, whereas if they are disjoint, the faces must not be merged. As a result, when this algorithm is implemented with floating-point arithmetic, it is unsound: when applied to edges that are almost touching but not quite, floating-point rounding error can cause the squares to be merged, even though they should not be. It is also not robust, because a seemingly negligible change in the position of the squares could change the decision of whether to merge.

Much effort has been devoted to approaches to improve the robustness of solid modeling and geometric algorithms [Brönnimann et al. 2001; Fang et al. 1993; Flórez et al. 2007; Fortune 1996; Guibas 1996; Hoffmann 1996; Hoffmann et al. 1988; Hu et al. 1996; Salesin et al. 1989; Sanjuan-Estrada et al. 2003; Schirra 1998; Sharma and Yap 2017; Snyder 1992a; Yap 1997]. These approaches vary in the shape representations used, types of numbers that may be represented (e.g., rational or algebraic numbers), the methods used, and the problems solved.

1.2 Soundness and Robustness

We formally define *soundness* and *robustness* in terms of pair of a *specification* of an ideal operation, $f : A \to B$, and computer *implementation* of that operation, `f : A -> B`, together with relations $\models_A \subseteq \mathtt{A} \times A$ and $\models_B \subseteq \mathtt{B} \times B$ relating ideal objects (A, B) to their representations (`A`, `B`, respectively).

We define an implementation `f` on representations to be *sound* if the representations can be related via $\models$ to ideal objects such that those ideal objects are related by f. We define an implementation `f` to be *robust* if it is a sound implementation of a specification defined by a continuous map.[1] Continuity implies that, in order to approximate an output to any finite level of precision, it suffices to inspect the input to a finite level of precision. For example, consider the representation of the result of $\sin(\pi)$ as a stream of ever finer intervals with rational endpoints:

```
# sin pi;;
[-4.6410206775e-8, 5.3589793242e-8]
[-8.0978056448535466350 7e-10, 1.21542010130123852029e-10]
[-5.01655761266833202356 2e-20, 4.04453249759190146481e-21]
...
```

We know $\sin(\pi) = 0$ and therefore soundness requires that each interval in the stream contains 0. Moreover, the computation is robust because the computation of each interval result of `sin pi`

[1] This definition is similar to Edalat and Lieutier [2002] who define a "robust algorithm" as "one whose correctness is proved with the assumption of a realistic machine model."

needs only an interval approximation of `pi`. This is in contrast to the unsoundness of floating-point computation that for many languages and CPUs returns `1.2246467991473532e-16`, which is not 0 and does not itself indicate a larger range of possible values that includes zero.

Accordingly, to reason about the soundness and robustness of CAD operations, the key starting point is to understand how to represent values – such as real numbers – and *shapes* – e.g., anything from simple lines, rectangles, and spheres, to complicated compositions thereof – as computational entities that can be soundly manipulated by computer programs.

1.3 Approach and Contributions

In this paper, we define several shape representations, define programs for sound and robust operations on them, and also demonstrate that some operations are discontinuous, and hence fundamentally not robust. Our approach is language-based in that we use our pure functional programming language, MarshallB, to develop StoneWorks, a library for sound and robust solid modeling.

We approach the representation problem through the lens of topology by identifying topological spaces whose *points* are shapes that have sound representations that we can naturally execute in MarshallB's topological semantics. For example, the stream of intervals in the `sin pi` example above illustrates the evaluation of $\sin(\pi)$ in the MarshallB REPL.

We have developed MarshallB by extending the Marshall language for exact real arithmetic [Bauer 2008] to include polymorphism, a Boolean datatype, and Riemannian integration. A key design of Marshall (and hence MarshallB) is that types denote topological spaces, functions denote *continuous maps*, and the language's semantics is sound with respect to these denotations. By developing a library for solid modeling in this language, we automatically ensure that all operations are sound and robust by the fact that all functions implemented in MarshallB are continuous.

We present the following contributions:

- We present MarshallB, an extension of Marshall with booleans and other operations designed for solid modeling (e.g, integration to compute the volume of a shape).
- We present *StoneWorks*, a library for solid modeling that exposes solid modeling operations on two shape representations: *O-rep*, a simplification of a well-known representation in solid modeling (i.e., *F-rep*), and *K-rep*, a novel representation that we have developed.
- We present a new representation for shapes, *compact representation* (*K-rep*), that enables specifying shapes and operations – such as Hausdorff distance and convex hull – that are not possible to compute for all shapes represented with other representations (i.e., F-rep).

We also present three modeling case studies that we've developed with StoneWorks:

1. We verify that the 3D mesh produced by a commercial CAD software system for a shape is sufficiently accurate by computing the Hausdorff distance between the mesh and the shape.
2. We produce a depth map for an image from a 3D scene represented by O-rep by implementing a robust ray intersection program.
3. We model a cam/piston system that has a parameterized angle of rotation and verify – as a computable analysis in the language – that the system is always separated from its enclosure for all possible rotation angles.

In total, StoneWorks demonstrates that it is possible to specify and execute solid modeling algorithms – soundly and robustly – in just a few lines of code. These programs can then serve as executable specifications to validate commercial or otherwise separately developed systems.

§2 demonstrates MarshallB and StoneWorks via a case study. §3 explains the syntax and semantics of MarshallB. §4, §5, §6, and §7 describe StoneWorks's shape representations, including K-rep. We finish with additional case studies (§8), related work (§9), and a conclusion (§10).

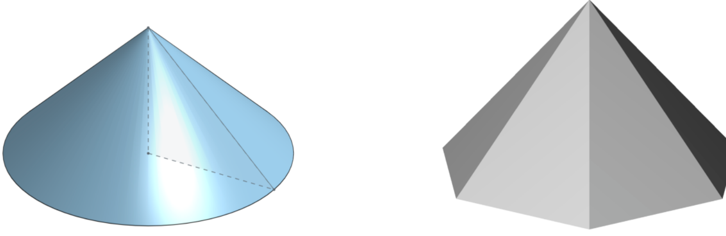

Fig. 1. *Left:* A cone (defined in Onshape). *Right:* A mesh that approximates the cone with 16 triangles (generated by Onshape).

```
let base : KShape (ℜ³) =
  map {ℜ²} {ℜ³} (λ xy : ℜ² ⇒ (xy[0], xy[1], 0)) unit_disk;;

let top : KShape (ℜ³) =
  compact_union {ℜ} unit_interval {ℜ³} (λ z : ℜ ⇒
  compact_union {ℜ²} (scale z unit_circle) {ℜ³} (λ xy : ℜ² ⇒
  point {ℜ³} (xy[0], xy[1], 1 - z)));;

let cone : KShape (ℜ³) =
 base `union3` top;;
```

Fig. 2. A MarshallB K-shape representing the ideal surface of a cone.

2 EXAMPLE: MESH VERIFICATION IN STONEWORKS

A solid modeling tool enables a user to model a solid and then generate an approximate polyhedral mesh. For example, Fig. 1 presents a cone together with an approximate mesh, both of which we generated using a CAD tool named Onshape. Such a mesh is useful for other end tasks, such as 3D printing, computer graphics, and finite element analysis.

Producing a mesh—or *meshing*—is a complex process for which soundness is a prominent concern [Nandi et al. 2018]. In general, one may choose the mesh density, which determines the size (and thus also number) of the polyhedra used in the approximation. Finer meshes are generally more accurate, but also more computationally expensive to produce and to process in applications that use the generated mesh. In general, it is difficult to assess the accuracy of a mesh with respect to the ideal shape, because often the ideal solid is not exactly representable in solid modeling tools.

In contrast, StoneWorks makes it possible to verify the accuracy of a generated mesh, bounding its Hausdorff distance from the ideal shape. Specifically, it is possible to encode in StoneWorks both the ideal surface of a cone and the polygonal approximation generated by Onshape and verify that their Hausdorff distance is at most a given value.

Like many solid modeling tasks, this verification task consists of first the *construction* of shapes—in this case, the exact cone and its polygonal approximation—followed by their *analysis*.

2.1 Construction

Fig. 2 presents MarshallB code that uses StoneWorks to encode the ideal surface of the cone. The cone is the union of its base with its top. The code represents each shape—the base, top, and together, the cone – by its *compact representation* in $\mathbb{R}^3$.

Compact Representation. A compact representation in $\mathbb{R}^3$ is a for-all quantifier `KShape` $\mathfrak{R}^3$ = ($\mathfrak{R}^3 \to \mathfrak{B}$) $\to \mathfrak{B}$ that, when applied to a predicate `P :` $\mathfrak{R}^3 \to \mathfrak{B}$, returns `tt` if `P` is true everywhere

```
let cvx3 (c : ℜ) (x y : ℜ³) : ℜ³ =
  (x[0] + c * (y[0] - x[0]), x[1] + c * (y[1] - x[1]), x[2] + c * (y[2] - x[2]));;

let triangle (x y z : ℜ³) : KShape (ℜ³) =
  compact_union {ℜ} unit_interval {ℜ³} (λ a : ℜ ⇒
  compact_union {ℜ} unit_interval {ℜ³} (λ b : ℜ ⇒ point {ℜ³} (cvx3 a x (cvx3 b y z))));;

let cone_mesh : KShape (ℜ³) =
  triangle (-5.31758e-17,1.0,-6.14666e-16) (-0.707107,0.707107,-7.11643e-16)
        (-1.2909e-16,0.0,1.0) `union3`
  ! ... 14 more triangles
  triangle (0.0,0.0,-2.39314e-16) (-0.707107,-0.707107,-7.11643e-16)
        (-1.0,0.0,-7.51812e-16);;
```

Fig. 3. A MarshallB K-shape representing the surface of a low-precision mesh generated for a cone.

in the shape and `ff` if `P` is false somewhere in the shape. For example, the unit interval on $\mathfrak{R}$ has the compact representation given by the following code:

```
let unit_interval : KShape ℜ = λ P : ℜ → 𝔅 ⇒
  mkbool (forall_unit_interval (λ x : ℜ ⇒ is_true  (P x)))
        (exists_unit_interval (λ x : ℜ ⇒ is_false (P x)));;
```

This code creates a *Belnap boolean* (Section 3.4) via `mkbool` using two predicates: one that verifies `P` to be true on the unit interval and one that computes if `P` is false on the unit interval. The Belnap boolean value evaluates to `tt` or `ff` depending on validity of each of its predicates, respectively. Both `forall_unit_interval` and `exists_unit_interval` are primitive quantifiers on $\mathfrak{R}$ made available by MarshallB and are naturally computable in MarshallB's semantics.

Simple Shapes. Building on `unit_interval`, StoneWorks additionally defines the simple shapes used in the code to define the cone. For example, StoneWorks's `map` operation enables any continuous *transformation* to be applied to a shape, such as translation, rotation, and scaling. In concert with `unit_interval`, StoneWorks can therefore provide a unit circle, defined as follows:

```
let unit_circle : KShape (ℜ²) =
  map {ℜ} {ℜ²}
    (λ t : ℜ ⇒ let theta = twopi * t in (cos theta, sin theta))
    unit_interval;;
```

StoneWorks's `compact_union` operation computes the union of a family of K-shapes that varies over points defined by another K-shape. In combination with `unit_interval` and `unit_circle`, StoneWorks can therefore provide a unit disk:

```
let unit_disk : KShape (ℜ²) =
  compact_union {ℜ} unit_interval {ℜ²} (λ r : ℜ ⇒ (scale r unit_circle));;
```

Generally, K-shapes form a monad representing nondeterminism (similar to the list monad), with the operations `point` and `compact_union` representing *return* and *bind*.

Cone. Using the these simple shapes, we can now build the cone. To build the base of the cone, the code simply maps the unit disk in $\mathfrak{R}^2$ to $\mathfrak{R}^3$. To build the top, the code leverages `compact_union` to union scaled and translated unit circles across the full height of the code. Finally, to assemble the full cone, the code uses StoneWorks's `union3` operator to union together the base

$$
\begin{aligned}
\text{expression } e &::= x \mid q \\
&\mid (e_1 \texttt{,} \ldots \texttt{,} e_n) \mid e[k] \\
&\mid \lambda\, x : t \Rightarrow e \mid e\ e \\
\text{type } t &::= \mathfrak{R} \mid \Sigma \mid t \texttt{*} \ldots \texttt{*} t \mid t \to t \\
\text{variable } &x \\
\text{rational } &q \\
\text{natural number } &k
\end{aligned}
$$

$$\frac{}{\Gamma \vdash q : \mathfrak{R}} \qquad \frac{\Gamma \vdash e_1 : t_1 \quad \ldots \quad \Gamma \vdash e_n : t_n}{\Gamma \vdash (e_1, \ldots, e_n) : t_1 \texttt{*} \ldots \texttt{*} t_n}$$

$$\frac{\Gamma \vdash e : t_1 \texttt{*} \ldots \texttt{*} t_n}{\Gamma \vdash e[k] : t_k} \qquad \frac{\Gamma, x : t_1 \vdash e : t_2}{\Gamma \vdash \lambda\, x : t_1 \Rightarrow e : t_1 \to t_2}$$

$$\frac{\Gamma \vdash e_1 : t_A \to t_B \qquad \Gamma \vdash e_2 : t_A}{\Gamma \vdash e_1\ e_2 : t_B}$$

Fig. 4. *Left*: Core syntax of MarshallB. *Right*: MarshallB typing rules.

and the top. StoneWorks's *set operations*, such as union, intersection, and relative complement allow the construction of composite shapes from simpler ones. This final step of modeling shapes using set operations is known as *Constructive Solid Geometry* (*CSG*) [Requicha and Voelker 1977].

Mesh. Constructing the mesh uses the same set of operations as constructing the cone (Fig. 3). The code constructs 16 triangles using `compact_union` and unions them together using `union3`.

2.2 Analysis

Upon constructing a shape, a developer often wants to *analyze* a shape in some way, for instance, to produce a rendering or make a measurement. In this example, we want to ensure that the Hausdorff distance between the ideal cone and its mesh is not too large.

The Hausdorff distance between two shapes is the maximum distance between a point on one shape and the closest point to it on the other shape. Using StoneWorks's `hausdorff_dist` operation we can define their Hausdorff distance by (where `d_R3 :` $\mathfrak{R}^3 \to \mathfrak{R}^3 \to \mathfrak{R}$)

```
let dist_between_cones = hausdorff_dist d_R3 cone cone_mesh
```

Note that `dist_between_cones` performs minimax optimization over uncountable infinitudes of real numbers, ranging over the points in each shape. This computational feat is possible because the two shapes are *compact* and the distance function, `d_R3`, between them is *continuous*, as all MarshallB functions are. Moreover, StoneWorks enables `dist_between_cones` to be evaluated to arbitrary precision. For example, we can compute that `dist_between_cones` is between 0.07 and 0.10.[2] Analytically, the Hausdorff distance between the two shapes is roughly (up to the floating-point discrepancies on the order of 10^{-16}) $1 - \cos\left(\frac{\pi}{8}\right) \approx 0.076$.

2.3 Summary

StoneWorks's construction on MarshallB – with continuity as a fundamental requirement of all programs – enables one to soundly and robustly construct a variety of shapes as well as reason about their composition. Over the next several sections, we present MarshallB, StoneWorks's two shape representations, and the constructions and analyses that StoneWorks exports.

3 SYNTAX AND TOPOLOGICAL SEMANTICS OF MARSHALLB

Fig. 4 presents MarshallB's core syntax and typing rules. MarshallB is an extension of System F with the primitive types $\mathfrak{R}$, Σ, and n-ary products. MarshallB also includes the primitive built-in constants shown in Fig. 5. Though not presented, MarshallB includes syntactic sugar for `let` expressions (that elaborate to lambdas) and top-level definitions, such as those of Coq or OCaml.

[2] Using a 3.1 GHz Core i7 (I7-7920HQ) CPU, `dist_between_cones < 0.1` returns `T` in 30 minutes, using 155 MB memory, and `dist_between_cones > 0.07` returns `T` in 36 minutes, using 398 MB memory.

```
⊤, ⊥ : Σ
(<) : ℜ → ℜ → Σ
(∧), (∨) : Σ → Σ → Σ
(+), (-), (*), (/) : ℜ → ℜ → ℜ
sin : ℜ → ℜ
dedekind_cut : (ℜ → Σ * Σ) → ℜ
forall_unit_interval : (ℜ → Σ) → Σ
exists_unit_interval : (ℜ → Σ) → Σ
exists_real : (ℜ → Σ) → Σ
integrate_unit_interval : (ℜ → ℜ) → ℜ
```

Fig. 5. MarshallB constants and their types. Infix operators are surrounded by parentheses.

In this section, we ground MarshallB's semantics in topology, on which we then build abstractions for real-valued arithmetic, logical operations, dedekind cuts, quantifiers, and integration. MarshallB also includes `sin` and other operators; however we omit their definitions due to space.

3.1 Denotational Semantics

The simply-typed lambda calculus can be interpreted into any cartesian closed category. We interpret MarshallB into the category of *presheaves over second-countable locally-compact sober topological spaces and continuous maps*, and thus interpret the primitive types as objects and primitive terms as arrows in this category.

Definition 3.1. A *second-countable locally-compact sober space* E is a tuple $(K, O, \sqsubseteq, V)$:[3]

- a countable poset K called a *compact basis*
- a countable set O called an *open sub-basis*
- a relation $\sqsubseteq \subseteq K \times O$, called a *containment relation*, that is upward-closed with respect to the K argument, i.e., if $k \sqsubseteq o$ and $k \le k'$, then $k' \sqsubseteq o$.
- a predicate $V \subseteq \mathcal{P}(O)$ on the powerset of the open basis, called a *validity predicate*.

For instance, MarshallB has a type $\mathfrak{R}$ that denotes a locally-compact sober space $[\![\mathfrak{R}]\!]$ (but we hereafter drop the brackets) whose *points* contains the reals ($\mathbb{R}$) as well as other exotic elements.

Compact Basis. The compact basis (or K-basis) of a space E—denoted by $K(E)$—is the space's set of primitive representations. For example, the compact basis of $\mathfrak{R}$ is the set of symbolic intervals $K(\mathfrak{R}) \triangleq \{[\underline{x}, \overline{x}] \mid \underline{x} \in \{-\infty\} \cup \mathbb{Q}, \overline{x} \in \mathbb{Q} \cup \{\infty\}\}$, partially ordered by reverse inclusion, $[\underline{x}, \overline{x}] \le [\underline{y}, \overline{y}]$ if $\underline{x} \le \underline{y}$ and $\overline{y} \le \overline{x}$. These represent partial information about a real number: the first component represents a rational lower bound (with $-\infty$ being a vacuous bound) and the second an upper bound (with ∞ vacuous).

Open Sub-basis. The open sub-basis, which we refer to by $O(E)$, represents the fundamentally observable properties of a space. The open sub-basis for $\mathfrak{R}$ is $O(\mathfrak{R}) \triangleq \{\cdot < q \mid q \in \mathbb{Q}\} \cup \{q < \cdot \mid q \in \mathbb{Q}\}$, the predicates describing being smaller or larger than some rational number.

Containment Relation. The containment relation $k \sqsubseteq o$ describes when the partial information described by k implies the observable property described by o. For $\mathfrak{R}$, $q < [\underline{x}, \overline{x}]$ (as shorthand for $[\underline{x}, \overline{x}] \sqsubseteq q < \cdot$) holds when $q < \underline{x}$, and $[\underline{x}, \overline{x}] < q$ holds when $\overline{x} < q$.

Validity Predicate. A validity predicate, $V(E)$, allows us to take subspaces of a larger space, E. For $\mathfrak{R}$, we take the whole space with $V(\mathfrak{R}) \triangleq \top$ because the additional elements (in comparison to $\mathbb{R}$) are frequently useful. For instance, `1 / 0` returns the constant stream $[-\infty, \infty]$. In general, all ground types in MarshallB have a validity predicate of $\top$.

Points. The *points* of a space E are monotone sequences of K-basis elements, $\mathbb{N} \to K(E)$ that *satisfy* the validity predicate $V(E)$ in that a sequence $x : \mathbb{N} \to K(E)$ is said to *lie* in a basic open $P \in O(E)$, written $x \models P$ if there is some $n \in \mathbb{N}$ such that $x_n \sqsubseteq P$. Moreover, the sequence x must satisfy $V(E)$, specifically $\{P \in O(E) \mid x \models P\} \in V(E)$. For example, the sequence that MarshallB returns for `sin pi` (Section 1.2) is a point that represents 0.

[3]Our definition of spaces differs from standard treatments and we feel that it may be helpful to point to other notions that motivated it, including [Taylor 2006], [Taylor 2019, Theorem 6.12], and [Vickers 1989, Chapter 9].

Sub-basic opens are positively observable in the sense that if a point lies in a sub-basic open, we can tell by looking at a finite prefix of the point's sequence, but the converse does not necessarily hold. Given points $x, y : \mathbb{N} \to E_K$, we say that y *specializes* x if every sub-basic open that x lies in, y lies in as well. Two points are considered equal when they lie in exactly the same sub-basic opens. That is, a point is uniquely determined by the basic opens it lies in. For instance, $x_n = [0, 0]$ and $y_n = [0, 1/n]$ are equal points of $\mathfrak{R}$ in that they both represent 0.

Continuous Maps. A *continuous map* $f : E \to_c F$ (we use the arrow $\to_c$ to distinguish from regular functions) is a monotone function (where $\mathbb{N}$ is given the standard numerical ordering) $K(f) : \mathbb{N} \times K(E) \to K(F)$ (but we will hereafter conflate f with $K(f)$), which when applied to points according to $(f(x))_n \triangleq K(f)_n(x_n)$, must map points of E to points of F, and moreover must map equal points of E to equal points of F. Two continuous maps are considered equal if they map input points to the same output points.

For example, a simple continuous map is negation, $(-) : \mathfrak{R} \to_c \mathfrak{R}$, which has the definition $(-)_n([\underline{x}, \overline{x}]) \triangleq [-\overline{x}, \underline{x}]$. Intuitively, a continuous map is a stream transformer that produces finite approximations of the output given only finite approximations of the input: this is the essence of continuity. A continuous map's monotonicity means that given more information about inputs and more time, functions must provide more information about outputs.

Product Spaces. There is a space $*$ with a single point: its compact basis $K(*)$ has one element and its open sub-basis $O(*)$ is empty. Points of E are isomorphic to continuous maps $* \to_c E$.

Finite products of ground types correspond to finite products of spaces with the product topology. The compact basis for the product space is the product of the spaces' compact bases. The open sub-basis for the product is the product of the open sub-bases.

Subspaces. As mentioned, $\mathfrak{R}$ contains more than the standard reals $\mathbb{R}$. However, we can define *subspaces* of a given space by defining additional properties that we require points of the space to satisfy. This is accomplished by specifying a nontrivial validity predicate that restricts which sequences of compact basis elements are valid points. For example, we can reason about subspaces of $\mathfrak{R}$, such as $\mathbb{R}$ itself, $\mathbb{R}$ along with $-\infty$ and ∞, and closed intervals of $\mathbb{R}$. The definitions of these subspaces result from validity predicates that compose the following properties of a point $x : \mathfrak{R}$.

(1) (Boundedness) There exist $\ell, u \in \mathbb{Q}$ such that $\ell < x$ and $x < u$
(2) (Disjointness) There is no $q \in \mathbb{Q}$ such that both $x < q$ and $q < x$.
(3) (Locatedness) For any $\ell, u \in \mathbb{Q}$ such that $\ell < u$, either $\ell < x$ or $x < u$.

With these properties, we can then define these subspaces as follows:

- $\mathbb{R}$: bounded, disjoint, and located points.
- $\overline{\mathbb{R}}$ ($\mathbb{R}$ along with $-\infty$ and ∞): disjoint and located points.
- $\mathbf{I}\mathbb{R}$ (closed intervals of real numbers $[a, b]$): bounded and disjoint points.

The topology on each of these spaces is the subspace topology inherited from $\mathfrak{R}$. Though these subspaces at not types within MarshallB, we refer to them later on in this paper.

Higher-order Functions. The Yoneda embedding lifts spaces into presheaves, and is full and faithful, and preserves products and exponentials when they exist. Accordingly, ground types denote spaces, and first-order functions denote continuous maps.

One need not think about presheaves until working with higher-order functions. Each higher-order primitive has a type `(A → B) → C` where `A`, `B`, and `C` are spaces (i.e., representable). A term of that type is thus equivalent to a function that takes continuous maps $\Gamma \times \mathtt{A} \to_c \mathtt{B}$ to maps $\Gamma \to_c \mathtt{C}$, for any representable Γ. In turn, this is just a map of stream transformers, from $\mathbb{N} \times K(\Gamma) \times K(\mathtt{A}) \to K(\mathtt{B})$ to $\mathbb{N} \times K(\Gamma) \to K(\mathtt{C})$. A special case of such a map is one whose $K(\mathtt{C})$-valued result on input (n, γ) always applies its input argument of type $\mathbb{N} \times K(\Gamma) \times K(\mathtt{A})$ to

$$[\![+]\!]_n([\underline{x},\overline{x}],[\underline{y},\overline{y}]) = [\underline{x}+\underline{y},\overline{x}+\overline{y}] \qquad [\![-]\!]_n([\underline{x},\overline{x}]) = [-\overline{x},-\underline{x}]$$
$$[\![*]\!]_n([\underline{x},\overline{x}],[\underline{y},\overline{y}]) = \mathsf{Kaucher}([\underline{x},\overline{x}],[\underline{y},\overline{y}])$$

Fig. 6. Semantics of arithmetic on $\mathfrak{R}$. Bauer and Taylor [2009, Section 12] define Kaucher multiplication.

$$[\![\top]\!]_n = \mathsf{True} \qquad [\![\bot]\!]_n = ? \qquad [\![\wedge]\!]_n(x,y) = \begin{cases}\mathsf{True} & x = \mathsf{True} \text{ and } y = \mathsf{True}\\ ? & \text{otherwise}\end{cases}$$

$$[\![<]\!]_n([\underline{x},\overline{x}],[\underline{y},\overline{y}]) = \begin{cases}\mathsf{True} & \overline{x} < \underline{y}\\ ? & \text{otherwise}\end{cases} \qquad [\![\vee]\!]_n(x,y) = \begin{cases}\mathsf{True} & x = \mathsf{True} \text{ or } y = \mathsf{True}\\ ? & \text{otherwise}\end{cases}$$

Fig. 7. MarshallB semantics of primitives on Σ.

the same (n, γ), and otherwise does not use its $K(\Gamma)$ argument, thus being equivalent to a map $\mathbb{N} \times (K(\mathsf{A}) \to K(\mathsf{B})) \to K(\mathsf{C})$.

3.2 Operational Semantics

The semantics as presheaves over spaces can be interpreted operationally (along the lines of, e.g., [Elliott 2017]). All the operations in constructing presheaves and describing spaces and continuous maps are fundamentally constructive. MarshallB only allows evaluation of closed terms whose types are representable, and accordingly, the categorical semantics of such a term is a point of the space represented by that type.

Operationally, a point is represented by a monotone sequence of K-basis elements. MarshallB computes this sequence, printing each K-basis element in sequence as it is computed. MarshallB computes ever finer estimates of the result, until the user is satisfied with a given approximation. (Operationally, continuous maps in MarshallB are simply *executable* stream transformers.) These operational semantics are *computationally sound* in that a computed sequence is a member of the equivalence class defining the point described by the program.

Note that these operational semantics of MarshallB that we present are only a simple pedagogical model for understanding the language's computational interfaces and computability; the actual runtime of MarshallB is more efficient than these operational semantics suggest. Bauer [2008] summarizes the runtime of Marshall, the language upon which MarshallB is based.

3.3 Real Arithmetic

Fig. 6 presents the semantics of standard arithmetic operators on $\mathfrak{R}$, such as negation, addition, and multiplication. As noted in Section 3.1 the K-basis for $\mathfrak{R}$ is the set of intervals $K(\mathfrak{R}) \triangleq \{[\underline{x},\overline{x}] \mid \underline{x} \in \{-\infty\} \cup \mathbb{Q}, \overline{x} \in \mathbb{Q} \cup \{\infty\}\}$. Therefore, arithmetic primitives perform rational interval arithmetic, propagating lower and upper bounds.

3.4 Logical Operations

Fig. 7 presents the semantics of primitives on Σ, the space of truth values called the *Sierpiński space*. The Sierpiński space allows us to represent a logical predicate U on a space E as an *open* subset of E (i.e., $U \subseteq E$) and therefore as a continuous map $\chi_U : E \to \Sigma$ that classifies which points lie in U. For example, the MarshallB expression `let i : ℜ → Σ = λ x : ℜ ⇒ 0 < x` describes the open subset of $\mathfrak{R}$ corresponding to the interval $(0, \infty)$.

An open subset or – *open* – is a predicate $U \subseteq E$ on points such that there is a corresponding predicate $U_K \subseteq K(E)$. The relationship between the two predicates is that $U(x)$ holds if and only if $\exists n \in \mathbb{N}.\ U_K(x_n)$. This relationship denotes that if a subset is open then it is possible to confirm that its predicate holds of a point by inspecting a finite approximation of that point, x_n.

This relationship does not entail that it is also possible to refute that an open holds of a point with a finite approximation. Therefore, an open – given by its corresponding continuous map – is *positively observable*: it is akin to a semidecision procedure that terminates on points in the open and diverges for points in its closed complement.

The Sierpiński space's K-basis is the partial order $\{?, \texttt{True}\}$, where $? \leq \texttt{True}$. It has a single basic open which holds of `True` but not ?. It has two points: $\top$, which is any stream that is eventually `True`, and $\bot$, which is the stream that is always ?.

Given an open U of E defined by a predicate U_K on the K-basis of E, the continuous map $\chi_U : E \to_c \Sigma$ that maps points in U to $\top$ and points in $E \setminus U$ to $\bot$ by

$$(\chi_U)_n(k) \triangleq \begin{cases} \texttt{True} & U_K(k) \\ ? & \text{otherwise} \end{cases}$$

MarshallB's comparison operator, `<` $: \mathfrak{R} \times \mathfrak{R} \to_c \Sigma$, instantiates this pattern on $\mathfrak{R}^2$ (Fig. 7), thereby providing comparison as a logical predicate in the language.

Continuous Logic. MarshallB also provides conjunctions and disjunctions of predicates. For example, the continuous map $\wedge : \Sigma \times \Sigma \to_c \Sigma$ conjoins predicates and demonstrates that opens are closed under intersection. For instance, the MarshallB expression `let i : ℜ → Σ = λ x : ℜ ⇒ 0 < x ∧ x < 1` describes the open subset of $\mathfrak{R}$ corresponding to the open interval $(0, 1)$.

The continuous map $\vee : \Sigma \times \Sigma \to_c \Sigma$ provides the disjunction of predicates and demonstrates that opens are also closed under union. In general, opens are also closed under infinitary disjunction (union): given continuous maps $f_i : E \to \Sigma$ for $i \in I$, where I is some indexing set, their union is

$$\left(\bigvee_{i:I} f_i\right)_n (x) \triangleq \begin{cases} \texttt{True} & \exists i \in I.\ f_{i\,n}(x) = \texttt{True} \\ ? & \text{otherwise} \end{cases}$$

In sum, opens are analogous to semidecidable languages: they are closed under finitary intersection and arbitrary union. However, just as with semidecidable languages, opens are not closed under complementation. Therefore, Σ does not admit negation. Instead to provide extended logical reasoning, MarshallB's provides a *quasi-boolean algebra* based on the *Belnap Booleans*.

Belnap Booleans. Fig. 8 defines MarshallB's $\mathfrak{B}$, the Belnap Booleans, and their corresponding logical operations, including negation. The Belnap Booleans form a quasi-Boolean algebra [Sherman et al. 2018] with corresponding logical operations. A Belnap Boolean is a pair of Σ values, denoted by `mkbool t f`, the first (`t`) indicating truth and the second (`f`) indicating falsity. Almost all of the laws that Boolean algebras satisfy also apply to these Boolean operators on $\mathfrak{B}$, with the exception that they do not necessarily satisfy the laws $x \vee \neg x = \top$ and $x \wedge \neg x = \bot$. For example, we have the counterexamples

$$\texttt{mkbool}\ \bot\ \bot\ \texttt{||}\ \neg\ (\texttt{mkbool}\ \bot\ \bot) = \texttt{mkbool}\ \bot\ \bot \neq \texttt{tt}$$
$$\texttt{mkbool}\ \top\ \top\ \texttt{\&\&}\ \neg\ (\texttt{mkbool}\ \top\ \top) = \texttt{mkbool}\ \top\ \top \neq \texttt{ff}.$$

Additionally, we can identify a few useful subspaces of $\mathfrak{B} = \Sigma \times \Sigma$:

$$\mathbb{B}_\bot \triangleq \{x \in \mathfrak{B} \mid \texttt{is_true x} \wedge \texttt{is_false x} = \bot\}$$
$$\mathbb{B} \triangleq \{x \in \mathbb{B}_\bot \mid \texttt{is_true x} \vee \texttt{is_false x} = \top\}$$

```
type 𝔅 = Σ * Σ;;
let mkbool (t f : Σ) : 𝔅  = (t, f);;
let is_true (x : 𝔅) = x[0];;
let is_false (x : 𝔅) = x[1];;

let (&&) (x y : 𝔅) : 𝔅 = mkbool (is_true  x ∧ is_true  y)
                                (is_false x ∨ is_false y);;
let (||) (x y : 𝔅) : 𝔅 = mkbool (is_true  x ∨ is_true  y)
                                (is_false x ∧ is_false y);;
let ¬ (x : 𝔅) : 𝔅 = mkbool (is_false x) (is_true x);;
let (<b) (x y : ℜ) : 𝔅 = mkbool (x < y) (y < x);;
let tt : 𝔅 = mkbool ⊤ ⊥;;
let ff : 𝔅 = mkbool ⊥ ⊤;;
```

Fig. 8. Definition of the Belnap Booleans and their operations in MarshallB.

$$[\![\texttt{dedekind_cut}]\!]_n(f) = \text{locate}(\text{bound}_n(1, f), f)$$

$$\text{bound}_0(b, f) = (0, ([-\infty, \infty])$$

$$\text{bound}_{n+1}(b, f) = \begin{cases} (n, [-b, b]) & \pi_1(f([-b, -b])) = \mathsf{True} \text{ and } \pi_2(f([b, b])) = \mathsf{True} \\ \text{bound}_n(2b, f) & \text{otherwise} \end{cases}$$

$$\text{locate}((0, [\ell, u]), f) = [\ell, u]$$

$$\text{locate}((n+1, [\ell, u]), f) = \text{locate}((n, [\ell', u']), f) \qquad \text{where}$$

$$a = \frac{2}{3}\ell + \frac{1}{3}u \qquad\qquad b = \frac{1}{3}\ell + \frac{2}{3}u$$

$$\ell' = \begin{cases} a & \pi_1(f([a, a])) = \mathsf{True} \\ \ell & \pi_1(f([a, a])) = \,? \end{cases} \qquad u' = \begin{cases} b & \pi_2(f([b, b])) = \mathsf{True} \\ u & \pi_2(f([b, b])) = \,? \end{cases}$$

Fig. 9. MarshallB semantics of Dedekind cuts.

The Booleans $\mathbb{B}$ represents the familiar discrete space with two points, while the space $\mathbb{B}_\perp$ has a third point `mkbool ⊥ ⊥`, which allows partiality. All of the Boolean operators in Fig 8 can be restricted to any of these spaces, e.g., the `||` of two points from $\mathbb{B}_\perp$ yields a point of $\mathbb{B}_\perp$.

One reason that we usually use 𝔅 rather than $\mathbb{B}$ is that there are few continuous maps to $\mathbb{B}$: there are no non-constant continuous maps into $\mathbb{B}$ from *connected* spaces, which include $\mathbb{R}^n$ and Σ. The map `<b : ℜ² → 𝔅` for comparing two real numbers restricts to a map $\mathbb{R}^2 \to_c \mathbb{B}_\perp$, and classifies whether a real number is smaller or larger than another, diverging only if they are equal.

3.5 Dedekind Cut

Fig. 9 presents the semantics of the primitive `dedekind_cut`, which describes a real number by a *Dedekind cut* `f : ℜ → Σ * Σ`, where `f(x)[0]` and `f(x)[1]` indicates whether the number is bigger or smaller than `x`, respectively. The primitive `dedekind_cut` equivalently has the type `(ℜ → 𝔅) → ℜ` and satisfies the identity `q <b dedekind_cut P = P q`, meaning that to describe a real number x with a Dedekind cut, we give a $\mathbb{B}_\perp$-valued predicate that indicates whether any test value $q \in \mathbb{R}$ is *less* than x.

```
let sqrt (x : ℜ) : ℜ = dedekind_cut (λ q : ℜ ⇒ q <b 0 || q² < x);;
```

n	2	sqrt 2	1	sqrt 2 + 1
0	$[2, 2]$	$[-\infty, \infty]$	$[1, 1]$	$[-\infty, \infty]$
1	$[2, 2]$	$[-\infty, \infty]$	$[1, 1]$	$[-\infty, \infty]$
2	$[2, 2]$	$[-2, 2]$	$[1, 1]$	$[-1, 3]$
3	$[2, 2]$	$[-2/3, 2]$	$[1, 1]$	$[1/3, 3]$
4	$[2, 2]$	$[2/9, 2]$	$[1, 1]$	$[11/9, 3]$
5	$[2, 2]$	$[22/27, 2]$	$[1, 1]$	$[49/27, 3]$
6	$[2, 2]$	$[98/81, 130/81]$	$[1, 1]$	$[179/81, 211/81]$

Fig. 10. Operational semantics of a program to compute $\sqrt{2} + 1$. Boolean operations are described in Fig. 8.

$$[\![\texttt{forall_unit_interval}]\!]_n(f) = \begin{cases} \texttt{True} & \forall i \in \{0, \ldots, 2^n - 1\}.\ f([i/2^n, (i+1)/2^n]) = \texttt{True} \\ ? & \text{otherwise} \end{cases}$$

$$[\![\texttt{exists_unit_interval}]\!]_n(f) = \begin{cases} \texttt{True} & \exists i \in \{0, \ldots, 2^n - 1\}.\ f([i+0.5)/2^n, (i+0.5)/2^n]) = \texttt{True} \\ ? & \text{otherwise} \end{cases}$$

$$[\![\texttt{exists_real}]\!]_n(f) = \begin{cases} \texttt{True} & \exists i \in \{0, \ldots, 2^{2n}\}. f([-2^n + i/2^n, -2^n + i/2^n]) = \texttt{True} \\ ? & \text{otherwise} \end{cases}$$

Fig. 11. MarshallB semantics of quantifiers.

Fig. 10 illustrates the operational semantics of a program to compute $\sqrt{2} + 1$; the program uses `dedekind_cut` to specify the value of $\sqrt{2}$. Dedekind cuts compute by first attempting to find *some* bound on the size of the number, by trying numbers of larger and larger magnitude until the predicate eventually says that it is smaller than the very positive number and larger than the very negative one. At step $n = 2$ in Fig. 10, the Dedekind cut for `sqrt` successfully bounds $\sqrt{2}$ to within $[-2, 2]$, thus bounding the overall result.

Upon bounding the number within some interval, the primitive proceeds to locate the number either in the lower two thirds of the interval or the upper two thirds of the interval. After $n = 2$, each successive step in evaluation runs locate to refine the interval in which $\sqrt{2}$ lies. Evaluation proceeds for $n > 6$ (not shown in the Figure) by producing finer and finer approximations of $\sqrt{2}$. Note that the more standard procedure of bisection could fail if the cut specifies a rational number exactly in the middle the interval, because the predicate would never recognize that test point as being either smaller or larger than the true value; this is why we use overlapping regions.

3.6 Quantifiers

Fig. 11 presents the semantics of quantifiers. The primitive `forall_unit_interval` and the primitive `exists_unit_interval` take a predicate `P : ℜ → Σ` on real numbers and indicate whether (or not) `P` holds everywhere on the unit interval $[0, 1] \subseteq \mathbb{R}$ or somewhere on [0,1], respectively. The primitive `exists_real` indicates whether a predicate holds for some real number. The universal quantifier `forall_unit_interval` iteratively partitions the unit interval into ever finer partitions, evaluating the predicate on these increasingly fine intervals. Its adequacy—in that whenever it is applied to any predicate `P` such that $\forall x \in [0, 1].\ \texttt{P}(x) = \top$, it eventually returns `True`—is a consequence of the fact that the unit interval on $\mathbb{R}$ is topologically compact. Existential quantifiers enumerate a dense collection of rational numbers, and return `True` when they encounter some rational number for which the approximated result is `True`.

$$[\![\texttt{integrate_unit_interval}]\!]_n(f) = \sum_{i=0}^{2^n-1} f([i/2^n, (i+1)/2^n]) \qquad \text{(sum using interval arithmetic)}$$

Fig. 12. MarshallB semantics of integration.

3.7 Integration

Fig. 12 presents the semantics of integration. The primitive `integrate_unit_interval` specifies the Riemannian integral of a function `f : ℜ → ℜ` over the unit interval [0,1]. Integration proceeds by partitioning the unit interval into ever finer partitions, approximating the integrand with rational interval-valued functions that are constant on each partition, whose interval-valued integrals can be found by simple finite sums. The integrals of these interval approximations will eventually converge as partitions are made finer.

4 STONEWORKS'S SOUND AND ROBUST REPRESENTATIONS

With MarshallB as its base, StoneWorks exposes and supports two alternative representations: *Open representation* (*O-rep*), which Edalat and Lieutier [2002] defines as the *solid domain*, is a canonicalization of *function representation* (F-rep). *Compact representation* (*K-rep*) is a novel representation in which a shape's representation is a functional that computes whether a predicate holds everywhere or somewhere within a shape. K-rep enables StoneWorks to implement and expose a variety of useful analyses that are not possible to expose for O-rep (or F-rep).

Open Representation. Open representation (O-rep) is a simplification of function representation (F-rep), a well-known shape representation for solid modeling. The F-rep of a shape $S \subseteq E$ is a function $f : E \to \mathbb{R}$ such that $f(x) > 0$ if x is in the interior of s, $f(x) < 0$ if x is in the exterior of s, and $f(x) = 0$ if x is on the boundary of s. F-rep is a natural specification for *implicit surfaces*: shapes that are specified by a mathematical equation whose zeros denote the shape's boundary. In contrast to B-rep (Section 1.1), F-rep can be used to implement sound and robust operations.

The O-rep of a shape S is the *unique* function $o : E \to \mathbb{B}_\bot$ such that $o(x) = \mathtt{tt}$ if x is in the interior of s, $o(x) = \mathtt{ff}$ if x is in the exterior of s, and $o(x) = \bot_\mathbb{B}$ if x is on the boundary of s. The topology on $\mathbb{B}_\bot$ ensures that this function o is continuous for any shape S.

O-rep is a quotient of F-rep, ensuring canonicity. However, all O-rep constructions and analyses that we present could just have well been done with continuous function representations.

Compact Representation. Our *compact repesentation*, or K-rep (K for compact), is an entirely new representation for shapes that describes a shape S by its universal quantification functional $k : (E \to \mathbb{B}_\bot) \to \mathbb{B}_\bot$: for any (continuous) predicate $P : E \to \mathbb{B}_\bot$ on points in the space E, $k(P)$ is $\mathtt{tt}$ if P is $\mathtt{tt}$ everywhere in S, $q(P)$ is $\mathtt{ff}$ if P is $\mathtt{ff}$ somewhere in S, and $\bot_\mathbb{B}$ otherwise.

K-rep enables the representation of shapes that are not representable in O-rep (and thus also F-rep), such as 2-dimensional shapes within 3-dimensional space. It also enables the computation of properties that are not generally possible with O-rep (or F-rep), including volume, Hausdorff distance, *separation distance* (infimum of closest points between two shapes), and nonemptiness [4].

Summary. Over the next several sections we detail StoneWorks's constructions and analyses for O-rep (Section 5), K-rep (Section 6), and *OK-shapes*, a shape with both an O-rep and K-rep (Section 7). OK-shapes admits analyses, such as volume, that are not available for a shape with only a single representation in isolation.

[4]Some of these properties can be computed with O-rep shapes given a priori knowledge that the shapes are within some compact space. We will later show that we can generate a K-representation from an O-representation when this is the case.

```
type OShape E = E → 𝔅
! Constructions
unit_interval_o : OShape ℜ
unit_disk_o, unit_square_o : OShape (ℜ²)
rectangle_o (width height : ℜ) : OShape (ℜ²)
product_o E F : OShape E → OShape F → OShape (E * F)
empty_o, full_o : (E : type) → OShape E
union_o, intersection_o : (E : type) → OShape E → OShape E → OShape E
complement_o E : OShape E → OShape E
contramap E F : (F → E) → OShape E → OShape F
translate_o_R2 (trans : ℜ²) : OShape (ℜ²) → OShape (ℜ²)
! Analyses
is_in E : E → OShape E → 𝔅
nonempty E (exists_E : (E → Σ) → Σ) : OShape E → Σ
```

Fig. 13. A module for open representation in MarshallB.

5 OPEN REPRESENTATION (O-REP)

Fig. 13 summarizes StoneWorks's definition of an *O-shape* (a shape given by its O-rep) and its associated constructions and analyses. Edalat and Lieutier [2002] defined O-rep and named such shapes "partial solids." We view this representation as important for programming with shapes within StoneWorks, particularly due to its fundamental interplay with compact representation.

In this section, we give *synthetic* versions of the many of Edalat and Lieutier [2002]'s results, in that we provide programs that witness the computability and continuity of their corresponding operation. This stands in contrast to Edalat and Lieutier [2002], who instead give semantic definitions of operations – without programs – and then provide proofs of computability and continuity (for particular semantic models).

5.1 Definition

First, we define some topological properties of shapes. Because opens admit arbitrary unions, for any property, there is a largest open subset satisfying that property. Given a subset $S \subseteq E$, its *interior* $\text{int}(S)$ is the largest open set of E contained in S, its *exterior* $\text{ext}(S)$ is the interior of the complement of S, and its *closure* $\overline{S}$ is the complement of the exterior of S. For instance, consider the half-open interval $[0, 1) \subseteq \mathbb{R}$. Its interior is $(0, 1)$, its exterior is $\{x \in \mathbb{R} \mid x < 0 \vee x > 1\}$, and its closure is $[0, 1]$.

Definition 5.1. An *O-shape* is a continuous map $E \to_c \mathbb{B}_\bot$.

That is, an O-shape $o : E \to_c \mathbb{B}_\bot$ is a partial map to the Booleans that indicates the shape's interior, which are the points x of E such that $o(x) = \mathtt{tt}$, and its exterior, which are those points x that are mapped to $\mathtt{ff}$. The map o diverges on points on the shape's boundary.

Definition 5.2. If an O-shape o satisfies $\{x : E \mid o(x) = \mathtt{tt}\} = \text{int}(O)$ and $\{x : E \mid o(x) = \mathtt{ff}\} = \text{ext}(O)$, then o is the *open representation* of O. Any O-shape o' specialized by o (classifying as $\mathtt{tt}$ or $\mathtt{ff}$ smaller areas in comparison to o) is called a *weak open representation* of O.

A shape has exactly one open representation, but may have many weak open representations. A weak open representation of O may classify some points in the interior or exterior of O with $\bot$, rather than a Boolean value. But if a weak open representation classifies a point as either $\mathtt{tt}$ or $\mathtt{ff}$, then that classification correctly indicates the point is the in interior or exterior, respectively.

Two shapes have the same O-representation if their interiors and exteriors are the same. For instance, both $\mathbb{Q}$ and $\mathbb{R} \setminus \mathbb{Q}$, as shapes in $\mathbb{R}$, have the same O-representation `λ x : 𝔑 ⇒ mkbool ⊥ ⊥`, since they both have empty interior and exterior.

5.2 Constructions

StoneWorks provides the unit interval, which is constructed in MarshallB as follows:

```
let unit_interval_o : 𝔑→𝔅 = λ x : 𝔑⇒ 0 <b x && x <b 1;;
```

Note that both the open unit interval $(0, 1)$ (as immediately seen above) and the closed unit interval $[0, 1]$ share the same O-rep and therefore this definition at once defines both intervals.

StoneWorks's unit disk $\{(x, y) : \mathbb{R}^2 \mid x^2 + y^2 \leq 1\}$ has an equally declarative implementation:

```
let unit_disk : 𝔑² →𝔅 = λ x : 𝔑² ⇒ x[0]² + x[1]² <b 1;;
```

StoneWorks supports Cartesian products of O-shapes, for instance enabling a user to easily specify the unit square as the product of two intervals:

```
let product_o E F (oE : OShape E) (oF : OShape F) : OShape (E * F) =
  λ x : E * F ⇒ oE x[0] && oF x[1];;

let unit_square : OShape (𝔑²) = product {𝔑} {𝔑} unit_interval_o unit_interval_o;;
```

5.2.1 *Constructive Solid Geometry (CSG) Operations.*

The Boolean operations on $\mathfrak{B}$ can be applied pointwise to yield Boolean operations on O-shapes, giving rise to the constructive solid geometry (CSG) operations of union, intersection, and complementation.

```
let union_o        E (o1 o2 : OShape E) : OShape E = λ x : E ⇒ o1 x || o2 x;;
let intersection_o E (o1 o2 : OShape E) : OShape E = λ x : E ⇒ o1 x && o2 x;;
let complement_o   E (o     : OShape E) : OShape E = λ x : E ⇒ ¬ (o x);;
```

These Boolean operations are only meaningful to the extent that they satisfy expected properties of Boolean operations. In general, O-shapes do not form a Boolean algebra: instead, the quasi-Boolean algebra structure of $\mathfrak{B}$ (described in §3.4) lifts to a quasi-Boolean algebra on O-shapes. Since the Boolean operations only form a quasi-Boolean algebra for O-shapes, rather than the Boolean algebra that they form (classically) for shapes, the Boolean operations on O-shapes do not necessarily preserve open representation. One law that quasi-Boolean algebras miss that Boolean algebras have is the law $x \vee \neg x = \top$, and this may indeed be violated for O-shapes. For instance, while the union of $\mathbb{Q}$ and $\mathbb{R} \setminus \mathbb{Q}$ is the entire space $\mathbb{R}$, the union of their O-representations, both `λ x : 𝔑 ⇒ mkbool ⊥ ⊥`, is still `λ x : 𝔑 ⇒ mkbool ⊥ ⊥`, while the O-representation of $\mathbb{R}$ is `λ x : 𝔑 ⇒ tt`. While open representations are not preserved, *weak* open representations are:

Proposition 5.3. *Boolean operations on O-shapes preserve* weak *open representation. E.g., if `a` and `b` are weak open representations of A and B, respectively, then `union_o a b` is a weak open representation of $A \cup B$.*

Proof sketch. We prove the case of union described above. Other cases are similar, or easier. We must show that if `is_true (a x || b x)`, then $x \in \text{int}(A \cup B)$, and if `is_false (a x || b x)`, then $x \in \text{ext}(A \cup B)$. If `is_true (a x || b x)`, then either `is_true (a x)` or `is_true (b x)`. Suppose the former case, w.l.o.g, and thus we know $x \in \text{int}(A)$, and thus $x \in \text{int}(A \cup B)$. If `is_false (a x || b x)`, then `is_false (a x)` and `is_false (b x)`, so $x \in \text{ext}(A)$ and $x \in \text{ext}(B)$, and thus $x \in \text{ext}(A \cup B)$. □

This weaker statement tells us that CSG operations will always yield O-shapes that, when they classify a point x as tt or ff, soundly indicate that the result of applying CSG operations on the underlying shapes also yields a shape such that x is in the interior or exterior, respectively. However, the O-shape may diverge on a point that is not on the boundary of the underlying shape. For instance, the intervals $[0, 1]$ and $[1, 2]$ are represented by the O-shapes `λ x : ℜ ⇒ 0 <b x && x <b 1` and `λ x : ℜ ⇒ 1 <b x && x <b 2`. While their union is $[0, 2]$, and $1 \in \mathrm{int}([0, 2])$, the union of their O-representations is

```
λ x : ℜ ⇒ (0 <b x && x <b 1) || (1 <b x && x <b 2)
```

which classifies 1 as `mkbool ⊥ ⊥` rather than `tt`.

5.2.2 Closure Under (Continuous) Inverse Images. Since `E ↦ E → 𝔅` is a contravariant functor [Edalat and Lieutier 2002] , we can take the preimage of any o-shape under a continuous map:

```
let contramap E F (f : F → E) (o : OShape E) : OShape F = λ x : F ⇒ o (f x);;
```

In particular, we can compute the *image* of any O-shape under a homeomorphism, since homeomorphisms have continuous inverses. Therefore, we can compute the image of O-shapes under translation and scaling by a nonzero factor, and for instance can create a rectangle from a square.

```
let translate_o_R2 (trans : ℜ^2) (o : OShape (ℜ^2)) : OShape (ℜ^2) =
  contramap {ℜ^2} {ℜ^2} (λ x : ℜ^2 ⇒ (x[0] - trans[0], x[1] - trans[1])) o;;

let rectangle_o (width height : ℜ) : OShape (ℜ^2) =
  contramap {ℜ^2} {ℜ^2} (λ x : ℜ^2 ⇒ (x[0] / width, x[1] / height)) unit_square;;
```

5.3 Analyses

As long as E is *overt*, meaning that it has an existential quantification functional of type `(E → Σ) → Σ` (as $\mathbb{R}^n$ does), we can affirm whether an O-shape has nonempty interior:

```
let nonempty E (exists_E : (E → Σ) → Σ) (o : OShape E) : Σ =
  exists_E (λ x : E ⇒ is_true (o x));;
```

By applying this operation after the Boolean operations, it is possible to affirm various properties, such as if a shape has nonempty exterior, or if the intersection of two shapes has nonempty interior, meaning that they overlap.

6 COMPACT REPRESENTATION (K-REP)

Fig. 14 summarizes StoneWorks's definition of *K-shapes* (a shape with a K-rep) and its provision of constructions and analyses for them. K-rep is our new shape representation that enables constructions and analyses not possible with O-rep (and thus also F-rep).

6.1 Definition

First, we define the topological notion of *compactness*: A subset $K \subseteq E$ is *compact* if for any space Γ and any open $U \subseteq \Gamma \times E$, the subset $\forall_K U \triangleq \{\gamma \in \Gamma \mid \forall x \in X.\ (\gamma, x) \in U\}$ is open in Γ.

A shape's compact representation is its $\mathbb{B}_\perp$-valued universal quantifier:

Definition 6.1. Let $k : (E \to_c \mathbb{B}_\perp) \to_c \mathbb{B}_\perp$ be a continuous map[5] and let K be a shape in E. If for all continuous maps $p : E \to_c \mathbb{B}_\perp$, $k(p) = \mathrm{tt}$ if and only if $\forall x \in K.\ p(x) = \mathrm{tt}$, and $k(p) = \mathrm{ff}$ if and only if $\exists x \in K.\ p(x) = \mathrm{ff}$, then k is the *compact representation* of K. We call k a *K-shape*.

[5] Here we let the $\to_c$ arrow represent the generalized notion of continuous map determined by the category of presheaves over spaces and continuous maps, so that higher-order functions are permissible. Equivalently, k is a function from maps $\Gamma \times E \to_c \mathbb{B}_\perp$ to $\Gamma \to_c \mathbb{B}_\perp$ for all Γ (natural in Γ).

```
type KShape E = (E → 𝔅) → 𝔅
! Constructions
unit_interval : KShape ℜ
unit_square : KShape (ℜ²)
unit_disk : KShape (ℜ²)
point E (x : E) : KShape E
compact_union E (k : KShape E) F (f : E → KShape F) : KShape F
product E F : KShape E → KShape F → KShape (E * F)
empty E : KShape E
union E (k1 k2 : KShape E) : KShape
intersect, difference : (E : type) → KShape E → OShape E → KShape E
map E F : (E → F) → KShape E) → KShape F
bezier E (cvx_comb : ℜ → E → E → E) (p0 p1 p2 : E) : KShape E
convex_hull E (cvx_comb : ℜ → E → E → E) : KShape E → KShape E
minkowski_sum E (plus : E → E → E) (k1 k2 : KShape E) : KShape E
! Analyses
forall   E : KShape E → (E → 𝔅) → 𝔅     ! synonym: is_contained_in
exists   E : KShape E → (E → 𝔅) → 𝔅     ! synonym: touches
forall_s, exists_s : (E : type) → KShape E → (E → Σ) → Σ
is_empty E : KShape E → 𝔅
infimum, supremum : KShape ℜ → ℜ
disjoint   E (neq : E → E → Σ) : KShape E → KShape E → Σ
kshape_neq E (neq : E → E → Σ) : KShape E → KShape E → Σ
separation_dist, hausdorff_dist : (E : type) → (E → E → ℜ) → KShape E → KShape E → ℜ
```

Fig. 14. A module for compact representation in MarshallB.

We use the MarshallB type `type KShape E = (E → 𝔅) → 𝔅` for K-shapes. Just as $\mathfrak{R}$ strictly contains $\mathbb{R}$ and $\mathfrak{B}$ strictly contains $\mathbb{B}$, `KShape E` strictly contains the K-shapes of E. For instance, `λ P : E → 𝔅 ⇒ mkbool ⊥ ⊥` is not a K-shape, because there is no shape that it represents.

A K-shape is defined *as* its universal quantifier on $\mathbb{B}_\perp$. Its existential quantifier can be defined by negation of the universal quantifier, owing to the fact that $\mathbb{B}_\perp$ behaves as a quasi-Boolean algebra:

```
let forall E (k : KShape E) (p : E → 𝔅) : 𝔅 = k p;;
let exists E (k : KShape E) (p : E → 𝔅) : 𝔅 = ¬ (k (λ x : E ⇒ ¬ (p x)));;
```

If we choose to think of the predicate P as an O-shape, then we can use `is_contained_in` and `touches` as synonyms for `forall` and `exists`, respectively.

Theorem 6.2. *Every shape in $\mathbb{R}^n$ that has a compact representation is closed and compact, and its compact representation is unique, meaning that K-shapes in $\mathbb{R}^n$ are in bijection with closed, compact subsets of $\mathbb{R}^n$.*

Proof. $\mathbb{R}^n$ is countably-based, locally compact, and sober, meaning that it is quasi-Polish. de Brecht and Kawai [2017] prove the correspondence for quasi-Polish spaces. □

Given any Σ-valued predicate on a space E, a K-shape on E can return a Σ-valued predicate indicating whether that predicate holds everywhere or somewhere on that shape:

```
let forall_s E (k : KShape E) (p : E → Σ) : Σ =
  is_true (k (λ x : E ⇒ mkbool (p x) ⊥));;

let exists_s E (k : KShape E) (p : E → Σ) : Σ =
  is_false (k (λ x : E ⇒ mkbool ⊥ (p x)));;
```

In fact, a (valid) pair of these quantifiers is isomorphic to a (valid) K-shape; one constructs a K-shape from these quantifiers with[6]

```
let from_quantifiers E (forall exists : (E → Σ) → Σ) : KShape E = λ P : E → 𝔅 ⇒
  mkbool (forall (λ x : E ⇒ is_true (P x))) (exists (λ x : E ⇒ is_false (P x)));;
```

6.2 Constructions

We can construct the closed unit interval $[0, 1]$ using logical language primitives:

```
let unit_interval : KShape ℜ = λ P : ℜ → 𝔅 ⇒
  mkbool (forall_unit_interval (λ x : ℜ ⇒ is_true  (P x)))
         (exists_unit_interval (λ x : ℜ ⇒ is_false (P x)));;
```

K-shapes form a (strong) monad and their monad operations are those inherited from the continuation monad `E ↦ (E → 𝔅) → 𝔅` of "generalized quantifiers" [Escardó and Oliva 2010]. It may also be viewed as a nondeterminism monad. This monad's unit gives the K-shape for a single point:

```
let point E (x : E) : KShape E = λ P : E → 𝔅 ⇒ P x;;
```

Common intuition for compact sets is that they behave much like finite sets. The monad's bind operation witnesses one such instance: the union of compactly many compact sets is compact:

```
let compact_union E (k : KShape E) F (f : E → KShape F) : KShape F
  = λ P : F → 𝔅 ⇒ forall {E} k (λ i : E ⇒ forall {F} (f i) P);;
```

We can use this to, for instance, construct a cone from disks that depend on a line segment, as seen in §2. Having a (strong) monad means that we also get products of K-shapes:

```
let product E F (kE : KShape E) (kF : KShape F) : KShape (E * F) =
  λ P : E * F → 𝔅 ⇒ forall {E} kE (λ x : E ⇒
                      forall {F} kF (λ y : F ⇒ P (x, y)));;
```

6.2.1 Constructive Solid Geometry (CSG) Operations. It is possible to compute disjunctive CSG operations on K-shapes:

```
let empty E : KShape E = λ P : E → 𝔅 ⇒ tt;;
let union E (k1 k2 : KShape E) : KShape E =
  λ P : E → 𝔅 ⇒ forall {E} k1 P && forall {E} k2 P;;
```

However, in general, it is not possible to compute the conjunctive operations. If the full space E is representable as a K-shape, then E is compact (which is not the case for $\mathbb{R}^n$). If intersection $(\cap)$ is possible on K-shapes in E, then E is *discrete* (also untrue of $\mathbb{R}^n$), meaning that there is a continuous map $(=) : E \times E \to_c \mathbb{B}$ classifying the diagonal:

```
let equal E (intersection : KShape E → KShape E → 𝔅) (x y : E) : 𝔅 =
  ¬ (is_empty (intersection (point x) (point y)));;
```

[6] This isomorphic type is effectively a presentation of the Vietoris powerspace [Johnstone 1982]. Johnstone [1982] describes the Vietoris construction for *locales*, which are a slightly modified notion of topological space. Translated closer to our language, the Vietoris powerspace for a space E is characterized as the space of points $(\Box, \Diamond) : (E \to \Sigma) \to \Sigma) \times (E \to \Sigma) \to \Sigma)$ (where all arrows here denote continuous maps), such that $\Diamond$ preserves arbitrary unions, $\Box$ preserves finitary intersections and directed unions, and for all opens P and Q of E,

$$\Box P \wedge \Diamond Q \leq \Diamond(P \wedge Q) \qquad \text{and} \qquad \Box(P \vee Q) \leq \Box P \vee \Diamond Q,$$

where the operators $\vee$, $\wedge$ applied to opens $E \to \Sigma$ represent the pointwise lifting of those operators on the Sierpiński space, and $U \leq V$ indicates that the open U is contained in the open V (where U and V are represented as continuous maps into Σ). The maps `forall_s` and `exists_s` correspond to $\Box$ and $\Diamond$ respectively.

The `is_empty` function, explained in §6.3, always returns a value in $\mathbb{B}$, and so the result is in $\mathbb{B}$ as well. Hence, intersection is not possible for spaces E of interest (such as $\mathbb{R}^n$), which are not discrete.

Nor are K-shapes closed under complementation, since, e.g., the complement of the empty space is the full space, which is not a K-shape unless the full space is compact.

While it is impossible to compute the intersection of two K-shapes, one can intersect an O-shape with a K-shape to produce a K-shape.

```
let intersect E (k : KShape E) (o : OShape E) : KShape E =
  λ P : E → 𝔅 ⇒ forall {E} k (λ x : E ⇒ ¬ (o x) || P x);;
```

The inner predicate can also be read as the Boolean implication `o x` implies `P x`: we are only required to satisfy `P` if we are within `o`. For this operation to work well, the O-shape o must be an *classical solid*, meaning that $\{x : E \mid f(x) \in \mathbb{B}\}$ is *dense*, i.e., its closure is the full space E [Edalat and Lieutier 2002]. Open shapes and closed shapes are classical solids, and classical solids are closed under the CSG operations [Edalat and Lieutier 2002]. All the usual shapes encountered in CAD or graphics applications are classical solids.

Theorem 6.3. *If o is a classical solid weakly representing O, and k is the K-representation of K, then* `intersect` *k o is the K-representation of $\overline{O} \cap K$, where $\overline{O}$ is the closure of O.*

Proof sketch. Suppose we have a predicate $\mathsf{P} : E \to_c \mathbb{B}_\perp$. We must confirm the two properties defining the K-representation of the output:

(1) $\forall x \in (\overline{O} \cap K).\ (\mathsf{P}(x) = \mathrm{tt})$ iff `is_true (forall k (λ x ⇒ ¬ (o x) || P x))`.
(2) $\exists x \in (\overline{O} \cap K).\ (\mathsf{P}(x) = \mathrm{ff})$ iff `is_false (forall k (λ x ⇒ ¬ (o x) || P x))`.

To show the property 1, we use the fact that O is a classical solid to equate $E \setminus \mathrm{ext}(O)$ with $\overline{O}$. Property 2 is fairly straightforward, depending only on some facts about point-set topology. □

We can use `intersect` to construct the K-shape for the closed unit disk using its open representation and the square $[-1, 1]^2$ (`square_sym`), for instance:

```
let unit_disk : KShape (ℜ²) = intersect {ℜ²} square_sym unit_disk_o;;
```

More generally, if we can produce a K-shape that bounds the O-representation of a closed shape O, we can produce a K-representation for O. In this way, `intersect` witnesses the topological theorem that a closed subset of a compact shape is compact.

Since O-shapes have complements, we can also do a relative complement of a K-shape and an O-shape:

```
let difference E (k : KShape E) (o : OShape E) : KShape E =
  intersect {E} k (complement_o {E} o);;
```

6.2.2 Closure Under (Continuous) Images. Since `E ↦ (E → 𝔅) → 𝔅` is a (covariant) functor, we can take the image of any K-shape under a continuous map:

```
let map E F (f : E → F) (k : KShape E) : KShape F =
  λ P : F → 𝔅 ⇒ forall {E} k (λ x : E ⇒ P (f x));;
```

The `map` program realizes the well-known result in topology that the continuous image of a compact set is compact. This allows us to compute projections, for instance from 3D to 2D. Of course, as with O-shapes, we can compute the image of shapes under homeomorphisms such as translation and scaling by a nonzero factor.

Closure under images also enables representation of parametric surfaces as K-shapes by first constructing the parameter space as a K-shape, and then taking its image under the parametric

definition. For instance, this permits construction of Bézier curves and surfaces (and NURBS, more generally). Given a space `E` with an function `cvx_comb : ℜ → E → E → E` for convex combinations, a quadratic Bézier curve is specified by

```
let bezier E (cvx_comb : ℜ → E → E → E) (p0 p1 p2 : E) : KShape E =
  map {ℜ} {E} (λ t : ℜ ⇒ cvx_comb t (cvx_comb t p2 p1) (cvx_comb t p1 p0))
              unit_interval;;
```

6.2.3 *Convex Hull and Minkowski Sum.*

Convex hulls can be computed with

```
let convex_hull E (cvx_comb : ℜ → E → E → E) (k : KShape E) : KShape E =
  compact_union {E} k {E} (λ x : E ⇒
  compact_union {E} k {E} (λ y : E ⇒
  map {ℜ} {E} (λ c : ℜ ⇒ cvx_comb c x y) unit_interval));;
```

Given two shapes S_1 and S_2 in a vector space E, their Minkowski sum is $S_1 \oplus S_2 \triangleq \{x + y \mid x \in S_1, y \in S_2\}$. Minkowksi sums may be used in CAD, for instance to round the edges of a solid by taking its Minkowski sum with a ball, as well as in robot path planning for determining a collision-free path-planning space.

When E is a vector space, it is possible to compute the Minkowski sum of K-shapes:

```
let minkowski_sum E (plus : E → E → E) (k1 k2 : KShape E) : KShape E =
  λ P : E → 𝔅 ⇒ forall {E} k1 (λ x : E ⇒
                forall {E} k2 (λ y : E ⇒ P (plus x y)));;
```

6.3 Analyses

A space E is *Hausdorff* if its codiagonal relation $\{(x, y) : E^2 \mid x \neq y\}$ is open. Note that $\mathbb{R}$ is Hausdorff, specified by the Marshall map `neq (x y : ℜ) : Σ = x < y ∨ y < x`. However, `ℜ` and $\mathbb{IR}$ are not Hausdorff. When `E` is Hausdorff with open codiagonal `neq : E → E → Σ`, we can compute the open exterior of a K-shape according to

```
let exterior E (neq : E → E → Σ) (k : KShape E) : E → Σ =
  λ x : E ⇒ forall_s {E} k (λ y : E ⇒ neq x y);;
```

The space of K-shapes gives us our first example of a non-discrete space that is disconnected: we can decide whether or not a compact shape is empty:

```
let is_empty E (k : KShape E) : 𝔅 = forall {E} k (λ x : E ⇒ ff);;
```

We can compute the minimum and maximum of nonempty K-shapes on $\mathbb{R}$:

```
let infimum (k : KShape ℜ) : ℜ =
  dedekind_cut (λ q : ℜ ⇒ forall {ℜ} k (λ x : ℜ ⇒ q <b x));;
let supremum (k : KShape ℜ) : ℜ =
  dedekind_cut (λ q : ℜ ⇒ exists {ℜ} k (λ x : ℜ ⇒ q <b x));;
```

Since K-shapes in a Euclidean space are bounded and closed, the extrema are finite, and they are obtained within the K-shape. Other notions of extrema on nonempty K-shapes over spaces other than $\mathbb{R}$ may be computed by first taking their image under an $\mathbb{R}$-valued continuous map.

If E is Hausdorff, we can affirm if two K-shapes in E are *disjoint*, meaning they have empty intersection:

```
let disjoint E (neq : E → E → Σ) (k1 k2 : KShape E) : Σ =
  forall_s {E} k1 (λ x : E ⇒ forall_s {E} k2 (λ y : E ⇒ neq x y));;
```

If E is a metric space with continuous distance metric $d : E \times E \to_c \mathbb{R}$, then we can compute the *separation distance*, i.e., the infimum of the distances between two K-shapes, as:

```
let separation_dist E (d : E → E → ℜ) (k1 k2 : KShape E) : ℜ =
  dedekind_cut (λ q : ℜ ⇒ q <b 0 || forall {E} k1 (λ x : E ⇒
                                       forall {E} k2 (λ y : E ⇒ q <b d x y)));;
```

If either shape is empty, the separation distance is ∞. If both shapes are nonempty, the separation distance is finite. In a metric space, two K-shapes are disjoint if and only if their separation distance is positive, `disjoint k1 k2 = 0 < separation_dist k1 k2`.

If E is a metric space with continuous distance metric $d : E \times E \to_c \mathbb{R}$, then we can compute the Hausdorff distance on K-shapes as:

```
let hausdorff_dist E (d : E → E → ℜ) (k1 k2 : KShape E) : ℜ =
  dedekind_cut (λ q : ℜ ⇒ q <b 0
    || exists {E} k1 (λ x : E ⇒ forall {E} k2 (λ y : E ⇒ q <b d x y))
    || exists {E} k2 (λ x : E ⇒ forall {E} k1 (λ y : E ⇒ q <b d x y)));;
```

If both shapes are empty, the Hausdorff distance is 0. If one shape is empty but the other is not, the Hausdorff distance is ∞. If both shapes are nonempty, the Hausdorff distance is finite.

For a metric space E, Hausdorff distance induces a metric on the non-empty K-shapes in E, and the topology induced by that metric is the Vietoris topology. If E is Hausdorff, then the space of K-shapes themselves are Hausdorff as well, and K-shapes are distinguished by the codiagonal:

```
let kshape_neq E (neq : E → E → Σ) (k1 k2 : KShape E) : Σ =
    exists_s {E} k1 (λ x1 : E ⇒ forall_s {E} k2 (λ x2 : E ⇒ neq x1 x2))
  ∨ exists_s {E} k2 (λ x2 : E ⇒ forall_s {E} k1 (λ x1 : E ⇒ neq x1 x2));;
```

In a metric space, two K-shapes differ if and only if their Hausdorff distance is positive, `kshape_neq k1 k2 = 0 < hausdorff_dist k1 k2`.

7 USING O-REP AND K-REP TOGETHER

The analyses available for O-shapes and K-shapes are complementary. For instance, we can test membership of a point in an O-shape (`is_in`), but this is not possible for K-shapes. On the other hand, analyses available for K-shapes, such as universal and existential quantification and separation and Hausdorff distance, are impossible for O-shapes. If we are able to represent the same shape with *both* open and compact representations, then analyses from *either* representation are available to us, and in fact, we can even perform new analyses not available with either representation, such as computing volume.

We call a shape with both a weak O-representation and K-representation an OK-shape. One might imagine that though we get the union of the analyses of O-rep and K-rep, we only get the intersection of their constructions. In a literal sense, yes, to construct an OK-shape, one must construct an O-shape and a K-shape. However, more constructions are available than one might expect. For instance, OK-shapes are closed under intersection, even though K-shapes are not; this is because, while K-shapes cannot be intersected with each other, a K-shape can be intersected with an O-shape to produce an O-shape.

Fig. 15 summarizes StoneWorks's definition of OK-shapes, and its provision of constructions and analyses for them. Semantically, OK-shapes correspond to the "bounded partial solids" of Edalat and Lieutier [2002]. While Edalat and Lieutier [2002] demonstrate continuity and computability of several of the constructions in this section, our synthetic approach that provides programs for

```
type OKShape E = KShape E * OShape E
! Constructions
unit_interval_ok : OKShape ℜ
unit_disk_ok : OKShape (ℜ^2)
product_ok E F : OKShape E → OKShape F → OKShape (E * F)
compact_union_ok E (i : KShape E) F (f : E → OKShape F) : OKShape F
empty_ok E : OKShape E
union_ok E (s1 s2 : OKShape E) : OKShape
intersect_ok, difference_ok : (E : type) → OKShape E → OShape E → OKShape E
bimap E F : (E → F) → (F → E) → OKShape E → OKShape F
minkowski_sum_ok E (plus : E → E → E) : OKShape E → KShape E → OKShape E
! Analyses
getK E : OKShape E → KShape E
getO E : OKShape E → OShape E
volume : OKShape E → ℜ
```

Fig. 15. A module for open-compact representation in MarshallB.

those constructions is new and quite different, especially since Edalat and Lieutier [2002] have no notion of K-representation on its own.

7.1 Definition

We consider an O-shape o and K-shape k to be *consistent* if o weakly represents the shape that k represents (recall that K-shapes represent shapes uniquely). We call consistent pair of an O-shape and K-shape an *OK-shape*.

7.2 Constructions

7.2.1 Closure Under Images of Homeomorphisms. Since `OShape` is contravariant in its space on which shapes are defined (hence closed under continuous preimages), and `KShape` is covariant (hence closed under continuous images), `OKShape` is invariant, implying that it is closed under *homeomorphisms*, which are continuous maps with continuous inverses (and include translation, rotation, and scaling by a nonzero factor).

7.2.2 Constructing O-Shapes Using K-Shapes. In this section, we witness some new constructions of O-shapes that are available using K-shapes. Analogous version of some of these constructions have already been shown for K-shapes, and putting them together allows us to have these constructions for OK-shapes as well.

The following programs witness another way in which compact sets behave like finite ones, in that "intersections of compactly many open sets are open" [Escardó 2009] (together with the less surprising fact that unions of compactly many open sets are open):

```
let compact_union_o E (k : KShape E) F (o : E → OShape F) : OShape F =
  λ x : F ⇒ exists {E} k (λ i : E ⇒ o i x);;

let compact_intersection_o E (k : KShape E) F (o : E → OShape F) : OShape F =
  λ x : F ⇒ forall {E} k (λ i : E ⇒ o i x);;
```

7.2.3 Constructive Solid Geometry (CSG) Operations. Since both O-shapes and K-shapes have the disjunctive CSG operations (`empty` and `union`), so do OK-shapes. OK-shapes are closed under intersection and relative complement as well, which might be surprising, since although O-shapes are

closed under these operations, K-shapes are not. However, one can produce the K-representation of the result by taking the intersection or relative complement of the K-representation of one shape with the O-representation of the other.

7.3 Analyses

It is possible to compute the volume of an OK-shape. First, we define an indicator function from $\mathbb{B}_\perp$ to $\mathbf{I}\mathbb{R}$

```
let indicator (b : 𝔅) : ℜ = dedekind_cut (λ q : ℜ ⇒ q <b 0 || (b && q <b 1));;
```

satisfying `indicator ff = 0`, `indicator tt = 1`, and $[\![$`indicator (mkbool ⊥ ⊥)`$]\!]_n = [0, 1]$. Volume on $\mathbb{R}$ is then defined as

```
let integral (a b : ℜ) (f : ℜ → ℜ) : ℜ =
  (b - a) * integrate_unit_interval (λ x : ℜ ⇒ f (a + x * (b - a)));;

let volume (s : OKShape ℜ) : ℜ = integral
  (infimum (getK s)) (supremum (getK s)) (λ x : ℜ ⇒ indicator (getO s x));;
```

If the boundary of s has finitely many points, then in fact volume(s) returns a real number rather than an interval. This can naturally be generalized to compute the volume of any OK-shape in $\mathbb{R}^n$, in which case the condition for the volume being a real number rather than an interval is that the boundary must have Jordan measure 0.

8 CASE STUDIES

We next demonstrate how MarshallB and StoneWorks enable the specification and execution of programs that solve several tasks in solid modeling.

8.1 Depth Map for O-Rep Using Root Finding for Ray Intersection

One technique to render computer graphics images is *ray tracing*: a procedure that computes the color of a pixel in an image by tracing the paths of the rays of light that propagate through the scene of interest, intersect with objects in the scene, and eventually intersect the pixel.

When shapes in a scene are represented with F-rep, computing the nearest point of intersection of a ray of light with the surface requires finding the smallest root of a function. For example, a sphere centered at $(3, -1, 2)$ with radius 5 would have the F-rep

$$f(x, y, z) = 5^2 - ((x - 3)^2 + (y + 1)^2 + (z - 2)^2).$$

To find the intersection of a ray that begins at the origin and moves in the $+x$-direction with this sphere, one needs to compute

$$\min\{x \in [0, \infty) \mid f(x, 0, 0) = 0\}.$$

A standard algorithm for this ray-intersection problem with black-box F-rep shapes is *ray marching*: choosing some fixed step size ε, walk along the ray with steps of size ε, and watch for the sign of f to change from negative to positive. This algorithm is not sound: choosing ε too large can cause the root to be missed, causing graphical artifacts; for instance, "thin features can disappear in some special surfaces" [Flórez et al. 2007]. As opposed to ray marching, StoneWorks enables a developer to provide a declarative specification of the intersection problem within a semantics that permits an execution using exact reals. Fig. 16 shows a short MarshallB program `lft_root` that performs robust ray intersection operations on O-shapes to produce a 2D image from a 3D scene. Its execution is similar to known approaches that use interval arithmetic for robust ray tracing of implicit surfaces [Flórez et al. 2007; Sanjuan-Estrada et al. 2003; Snyder 1992a]. Fig. 17 uses

```
let interval (l : ℜ) : KShape ℜ =
  map {ℜ} {ℜ} (λ x : ℜ ⇒ x * l) unit_interval;;

let lft_root (f : ℜ → 𝔅) : ℜ =
  dedekind_cut (λ q : ℜ ⇒ q <b 0 || ¬ (touches {ℜ} (interval q) f));;

let lft_root_max_20 (f : ℜ → 𝔅) : ℜ =
  dedekind_cut' [0, 20] (λ q : ℜ ⇒ ¬ (touches {ℜ} (interval q) f));;

let ray (disp : ℜ²) : ℜ → ℜ³ = let mag = sqrt (1 + disp[0]² + disp[1]²) in
  λ t : ℜ ⇒ (t / mag, t * disp[0] / mag, - t * disp[1] / mag);;

let ray_depth (scene : ℜ³ → 𝔅) (disp : ℜ²) : ℜ =
  lft_root_max_20 (λ t : ℜ ⇒ scene (ray disp t));;
```

Fig. 16. `lft_root` computes ray intersection. `ray_depth`, when given a 3D scene, describes the distance to the nearest point in the scene as a function of the 2D image coordinates. As an optimization, for producing images, we use `lft_root_max_20` rather than `lft_root` in the definition of `ray_depth`, since beyond this will appear black in the image. For any `f`, `dedekind_cut' [a, b] f = dedekind_cut (λ q : ℜ ⇒ q < a || (f q && q < b))` but the former is more performant. It would be possible to implement a compiler optimization that applies this operation on any definition `max k d` where `k` is constant and the normal form of `d` is a Dedekind cut, but we have yet to do so.

`lft_root` to create a 2D depth map image (`ray_scene`) of a 3D scene of a ball on a table, where the image's output is displayed for the rectangle $[-1, 1]^2$.[7]

Fundamentally, `lft_root` computes in a similar way to related work using interval arithmetic for ray intersection. Let's observe how `lft_root` behaves with some examples, before formally characterizing it. If we pass a ray through the half-line `λ x : ℜ ⇒ k <b x`, for some $k \in (0, \infty)$, the `lft_root` result is k. If we pass a ray through the empty shape, `lft_root (λ x : ℜ ⇒ ff)`, there is no intersection, and accordingly the result is ∞. A ray intersects the full shape immediately, i.e., `lft_root (λ x : ℜ ⇒ tt)` is 0. Tangency is problematic. For instance, taking $k \in (0, \infty)$, the result for the shape `λ x : ℜ ⇒ (x - k)² <b 0` is the point y such that $q < y$ if $q < k$ but $y \not< q$ for all $q \in \mathbb{Q}$., which is not located. `lft_root` satisfies:

$$\begin{aligned} q < \texttt{lft_root}(f) &\iff \forall x \in [0, q].\ f(x) = \mathrm{ff} \\ \texttt{lft_root}(f) < q &\iff \exists x \in [0, q].\ f(x) = \mathrm{tt} \end{aligned}$$

Fix an f and let us define $y = \texttt{lft_root}(f)$. Note that if f outputs only to $\mathbb{B}_\perp$, then y is always disjoint. If there is some $x \in [0, \infty)$ such that $f(x) = \mathrm{tt}$, then the cut is also bounded ($y < u$ for some u); if not, the result is ∞. The resulting cut will be located if for every $\ell, u \in \mathbb{Q}$ such that $\ell < u$, we have

$$\forall x \in [0, \ell].\ f(x) = \mathrm{ff} \quad \text{or} \quad \exists x \in [0, u].\ f(x) = \mathrm{tt}.$$

Intuitively, this will happen if f (assuming f outputs only to $\mathbb{B}_\perp$) switches away from ff, and when it does so it switches to tt after passing through `mkbool ⊥ ⊥` only at an isolated point.[8] This isolated point is $\texttt{lft_root}(f)$.

`lft_root` can be interpreted as realizing an abstracted version of the *constructive intermediate value theorem* [Taylor 2010].

[7] Producing this 256x256-pixel depth map takes 2 hours and 25 minutes, using 14 MB memory. We believe it would be possible to add optimizations that would reduce the runtime.
[8] This is analogous to the notion of a *stable zero* given by Taylor [2010].

```
let lta (x : ℜ) (y : ℜ) : 𝔅 =
  mkbool (x < y + 0.01) (y < x);;
let table (x : ℜ³) : 𝔅 =
  lta x[2] (-1) && lta (-1) x[1] && lta x[1] 1;;
let ball (x : ℜ³) : 𝔅 =
  lta ((x[0] - 2)² + (x[1] - 0.7)² + x[2]²) 1;;

let ray_scene : ℜ² → ℜ =
  ray_depth (union_o {ℜ³} table ball);;
```

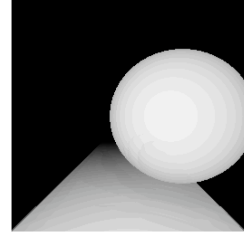

Fig. 17. `ray_scene` describes the distance to the nearest point in the ball-table scene as a function of the 2D image coordinates. The table and ball are defined with approximate comparisons, yielding thin regions where the interior and exterior overlap, which facilitates quick computation with a known maximal degree of inaccuracy. *Right:* A 256x256-pixel rendering of the `ray_scene` depth map rendered with MarshallB.

```
let piston : KShape (ℜ³) = compact_union {ℜ} (line_segment 20) {ℜ³} (λ x : ℜ ⇒
  compact_union {ℜ²} unit_disk {ℜ³} (λ yz : ℜ ⇒
  point (x, yz[0], yz[1])));;

let cam (angle : ℜ²) : KShape (ℜ³) =
  rotate_xy angle (translate_R3 (-2.5, 0, 0) (extrude 1 (scale2 5 unit_disk)));;

let translated_piston (angle : ℜ²) : KShape (ℜ³) =
  translate3 (5 + 2.5 * angle[0], 0, 0) piston;; !piston translates when cam rotates

let cam_piston (angle : ℜ²) : KShape (ℜ³) =
  union {ℜ³} (cam angle) (translated_piston angle);;

let enclosure_piece : KShape (ℜ³) = translate3 (30, -1, 0) (box (2, 2, 2));;

let collision_safe : Σ = forall_s {ℜ²} unit_circle (λ angle : ℜ² ⇒
  disjoint {ℜ³} neq_R3 (cam_piston angle) enclosure_piece);;

let clearance_dist : ℜ = infimum (map {ℜ²} {ℜ}
  (λ angle : ℜ² ⇒ separation_dist {ℜ³} d_R3 (cam_piston angle) enclosure_piece)
  unit_circle);;
```

Fig. 18. StoneWorks code that defines a cam/piston system that varies with the angle that the cam is at and an enclosure piece nearby the cam/piston system. `collision_safe` indicates whether or not the two parts are always separated, and then `clearance_dist` computes the separation distance between the two parts.

8.2 Implementing and Verifying a Cam-Piston System with StoneWorks

Verifying minimum clearance of a system with degrees of freedom is an important property for many designs. Commercial CAD software does not allow a user to do so in an automated way. Instead, the user must manually move or manipulate an object to sweep through its degrees of freedom to observe its clearance with respect to other objects. Either numerical error in the clearance computation, or failure to sufficiently test all possible configurations of the system, may cause the clearance computation to be unsound.

We will use StoneWorks to implement a model of a mechanical system that involves an eccentric cam driving a piston within an enclosure, to compute whether the cam and piston remain disjoint from the enclosure, and then to compute the minimum distance between the cam-piston system

and its enclosure. Fig. 18 shows the code for this task. StoneWorks automatically computes that, in this case, the system is indeed safe from collisions, as well as the minimum clearance distance between the components.

Construction. The cylindrical `piston` is defined by *extruding* the 2D disk `unit_disk : KShape` $(\mathfrak{R}^2)$ into a cylinder. Similarly, the `cam` is an extruded ellipse. When the cam rotates, the piston will translate along its axis of movement, as captured by `translated_piston`. We will want to consider the cam and piston system together when determining whether it collides with its enclosure, so we compute their union in the definition of `cam_piston` in Fig. 18. The `unit_circle` is used to parameterize the the possible rotations of the cam.

Analysis. We want to ensure that the cam-piston system of Fig. 18 is separated from its enclosure, regardless of its angle of rotation, and then to see how much clearance it has. For each possible rotation angle, we use `disjoint` to compute whether the cam and piston are disjoint from their enclosure, producing a Σ-valued predicate parameterized by the unit circle (where each point on the circle represents an angle). The `collision_safe` quantifies over all angles for the cam (drawn from the `unit_circle`) to compute whether the shapes are disjoint for all rotations. In this example, `collision_safe` returns `T`,[9] meaning that the two parts are always disjoint.

This implies that the `clearance_dist` will be a positive real number. Note that `collision_safe` verifies safety holds for an uncountable infinitude of possible angles of the circle, and `clearance_dist` computes the maximum of an uncountable infinitude of real numbers; this computational feat is possible because the circle is compact, and the functions that are computed over the circle in each case are continuous.

9 RELATED WORK

The syntax, semantics, and implementation of MarshallB are largely derived from the Marshall programming language [Bauer 2008].

Edalat and Lieutier [2002] use domain theory to develop a theory of computability of CAD operations. They define *partial solids*, which correspond closely to our O-shapes, and demonstrate their closure under Boolean operations. Their *bounded partial solids* correspond closely to our OK-shapes, and demonstrate computability of Minkowski sum. Our notion of K-shapes is new relative to Edalat and Lieutier [2002], including those operations that are possible on K-shapes but not OK-shapes, like the functorial and monadic operations, including representation of parametric surfaces like Bézier curves. For instance, there is no way to represent 2D surfaces in 3D space and compute their Hausdorff distance, as is done in §2. They do not present a programming model, nor do they implement any of the operations that they demonstrate are computable.

Various works study notions of robustness in computational geometry and approaches to achieve that [Brönnimann et al. 2001; Fang et al. 1993; Fortune 1996; Guibas 1996; Hoffmann 1996; Hoffmann et al. 1988; Salesin et al. 1989; Schirra 1998; Sharma and Yap 2017; Yap 1997]. Some approaches consider the use of approximate primitives that have error guarantees. In general, these guarantees are not compositional, so algorithms are considered which *can* provide guarantees using these approximate primitives [Hoffmann et al. 1988].

Some approaches (e.g., [Burnikel et al. 1995]) consider the use of exact arithmetic on algebraic numbers, which notably have decidable equality. CGAL [Fabri et al. 2000] follows this approach, offering geometric algorithms that operate parametrically over number types. The interface for these number types requires total (decidable) ordering, which necessarily limits the class of numbers that can be exactly represented, for instance precluding MarshallB's $\mathbb{R} \subseteq \mathfrak{R}$. As the constructible numbers are algebraic, many geometric computations can be performed with only these, though

[9] Using a 3.1 GHz Core i7 (I7-7920HQ) CPU, `collision_safe` returns `T` in 0.4 seconds, using 8 MB memory.

Sharma and Yap [2017] note that “Non-algebraic computation over Ω_4 [the elementary numbers] is important in practice”. Our framework for the analytical real numbers admits such computations.

The *Soft Exact Computation* paradigm in computational geometry [Sharma and Yap 2017] considers similar computational models and guarantees to this work: numerical computation “with a priori correctness guarantees, and which can dynamically adjust their arbitrary precision and steps depending on the input instance.” The *Subdivision Model* uses interval arithmetic for computing on shapes. This model has been applied to ray intersection in graphics [Flórez et al. 2007; Sanjuan-Estrada et al. 2003] as well as meshing [Lin and Yap 2011; Plantinga and Vegter 2004]. In all of these cases, computations are performed on F-rep shapes. In comparison, our definition of K-representation and computations on K-shapes is novel.

GENMOD [Snyder 1992b] is a programming system for “generative modeling” of shapes that represents shapes as parametric surfaces, generally based on hyper-rectangular domain. Using interval arithmetic, GENMOD provides two generic higher-order algorithms, “SOLVE, which computes solutions to a system of constraints, and MINIMIZE, which computes the global minimum of a function, subject to a system of constraints” [Snyder 1992a]. These are used for “ray tracing, computation of toleranced polygonal decompositions, detection of collisions, computation of CSG operations, approximation of silhouette curves, and many others.” A GENMOD parametric representation (called a *manifold*) is a map $\mathbb{R}^m \to \mathbb{R}^k$, usually implicitly assuming a hyper-rectangular domain within the input space $\mathbb{R}^m$, but the domain cannot be programmatically associated with the manifold. Thus GENMOD manifolds are not closed under union, for instance. GENMOD manifolds can be interpreted as K-shapes in MarshallB, but K-shapes are strictly more general. For instance, there are no GENMOD operators analogous to `point`, `union`, or `intersect`.

dReal is a tool that allows computation of approximate truth values over $\mathbb{R}$ [Gao et al. 2012], allowing order comparisons and bounded quantifiers. This tool allows the approximate computation of geometric facts in a manner similar to our tool.

Nandi et al. [2018] describe a simple CSG-based functional programming language for polyhedra, and gives a compiler correctness proof for compilation to meshes. Their language only admits polyhedral shapes, and their compiler does not consider computability on the reals: they use a primitive intersect that indicates whether a ray intersects a face, which is discontinuous, and hence not computable in the manner we discuss here.

Simpson [1998] presents exact real functionals for maximizing and integrating real-valued functions within a lazy functional programming language, making critical use of laziness in the operational execution of the functions to maximize or integrate.

10 CONCLUSION

Solid modeling is a challenging problem for which researchers have long sought sound and robust systems. StoneWorks provides an approach that is grounded in an underlying programming language for continuous computation, MarshallB. In combination with our novel compact representation, StoneWorks makes it possible to compute metrics, such as Hausdorff distance, that (to the best of our knowledge) were not possible to compute before with this level of generality.

In sum, our work holds out the promise of future language-based solutions for solid modeling that offer soundness and robustness guarantees as well as powerfully general programs for constructing and analyzing shapes.

ACKNOWLEDGMENTS

We thank Eric Atkinson, Thomas Bourgeat, Tej Chajed, and Alexander Renda for their feedback on earlier drafts. We thank Elijah Miller and David Kaufman for productive discussions about solid modeling. This work was supported by the Office of Naval Research (ONR N00014-17-1-2699).

REFERENCES

Andrej Bauer. Efficient computation with Dedekind reals. In *International Conference on Computability and Complexity in Analysis*, 2008.

Andrej Bauer and Paul Taylor. The Dedekind reals in Abstract Stone Duality. *Mathematical structures in computer science*, 19(4):757–838, 2009.

Hervé Brönnimann, Christoph Burnikel, and Sylvain Pion. Interval arithmetic yields efficient dynamic filters for computational geometry. *Discrete Applied Mathematics*, 109(1-2):25–47, 2001.

Christoph Burnikel, Jochen Könemann, Kurt Mehlhorn, Stefan Näher, Stefan Schirra, and Christian Uhrig. Exact geometric computation in LEDA. In *Symposium on Computational geometry*, 1995.

Matthew de Brecht and Tatsuji Kawai. On the commutativity of the powerspace constructions. *arXiv preprint arXiv:1709.06226*, 2017.

Abbas Edalat and André Lieutier. Foundation of a computable solid modelling. *Theoretical Computer Science*, 284(2):319–345, 2002.

Conal Elliott. Compiling to categories. In *ICFP*, 2017.

Martın Escardó. Intersections of compactly many open sets are open. 2009. Available at http://www.cs.bham.ac.uk/~mhe/papers/compactness-submitted.pdf.

Martin Escardó and Paulo Oliva. What sequential games, the Tychonoff theorem and the double-negation shift have in common. In *Workshop on Mathematically structured functional programming*, 2010.

Andreas Fabri, Geert-Jan Giezeman, Lutz Kettner, Stefan Schirra, and Sven Schönherr. On the design of CGAL, a computational geometry algorithms library. *Software: Practice and Experience*, 30(11):1167–1202, 2000.

Shiaofen Fang, Beat Bruderlin, and Xiaohong Zhu. Robustness in solid modelling: a tolerance-based intuitionistic approach. *Computer-Aided Design*, 25(9):567–576, 1993.

Jorge Flórez, Mateu Sbert, Miguel A Sainz, and Josep Vehí. Efficient ray tracing using interval analysis. In *International Conference on Parallel Processing and Applied Mathematics*, 2007.

Steven Fortune. Robustness issues in geometric algorithms. In *Applied Computational Geometry Towards Geometric Engineering*, pages 9–14. Springer, 1996.

Sicun Gao, Jeremy Avigad, and Edmund M Clarke. Delta-decidability over the reals. In *LICS*, 2012.

Leonidas J Guibas. Implementing geometric algorithms robustly. In *Applied Computational Geometry Towards Geometric Engineering*, pages 15–22. Springer, 1996.

Christoph M Hoffmann. How solid is solid modeling? In *Applied Computational Geometry Towards Geometric Engineering*, pages 1–8. Springer, 1996.

Christoph M Hoffmann, John E Hopcroft, and Michael S Karasick. Towards implementing robust geometric computations. In *Symposium on Computational geometry*, 1988.

Chun-Yi Hu, Nicholas M Patrikalakis, and Xiuzi Ye. Robust interval solid modelling part 1: representations. *Computer-Aided Design*, 28(10):807–817, 1996.

Peter T Johnstone. The Vietoris monad on the category of locales. In *Continuous lattices and related topics*, volume 27, pages 162–179. University of Bremen, 1982.

Long Lin and Chee Yap. Adaptive isotopic approximation of nonsingular curves: the parameterizability and nonlocal isotopy approach. *Discrete & Computational Geometry*, 45(4):760–795, 2011.

Chandrakana Nandi, James R Wilcox, Pavel Panchekha, Taylor Blau, Dan Grossman, and Zachary Tatlock. Functional programming for compiling and decompiling computer-aided design. In *ICFP*, 2018.

Simon Plantinga and Gert Vegter. Isotopic approximation of implicit curves and surfaces. In *Eurographics/ACM SIGGRAPH symposium on Geometry processing*, 2004.

A.A.G. Requicha and H.B. Voelker. *Constructive Solid Geometry, TM-25*. Technical memorandum. Production Automation Project, University of Rochester, 1977.

D Salesin, J Stolfi, and L Guibas. Epsilon geometry: Building robust algorithms from imprecise computations. In *Symposium on Computational Geometry*, 1989.

JF Sanjuan-Estrada, Leocadio G Casado, and Inmaculada García. Reliable algorithms for ray intersection in computer graphics based on interval arithmetic. In *Brazilian Symposium on Computer Graphics and Image Processing*, 2003.

Stefan Schirra. Robustness and precision issues in geometric computation. 1998.

Vikram Sharma and Chee K. Yap. Robust geometric computation. In J.E. Goodman, J. O'Rourke, and C. D. Tóth, editors, *Handbook of Discrete and Computational Geometry*, chapter 45, pages 1189–1223. CRC Press, 3 edition, 2017.

Benjamin Sherman, Luke Sciarappa, Adam Chlipala, and Michael Carbin. Computable decision making on the reals and other spaces: via partiality and nondeterminism. In *LICS*, 2018.

Alex K Simpson. Lazy functional algorithms for exact real functionals. In *International Symposium on Mathematical Foundations of Computer Science*, pages 456–464. Springer, 1998.

John M Snyder. Interval analysis for computer graphics. *ACM SIGGRAPH Computer Graphics*, 26(2):121–130, 1992a.

John M Snyder. *Generative Modeling for Computer Graphics and CAD: Symbolic Shape Design Using Interval Analysis*. Academic Press Professional, Inc., 1992b.

Paul Taylor. Computably based locally compact spaces. *Logic Methods in Computer Science*, 2:1–70, 2006.

Paul Taylor. A lambda calculus for real analysis. *Journal of Logic and Analysis*, 2:1–115, 2010.

Paul Taylor. Local compactness and bases in various formulations of topology. 2019. URL https://www.paultaylor.eu/ASD/loccbv/.

Steven Vickers. *Topology via logic*. Cambridge University Press, 1989.

Chee-Keng Yap. Towards exact geometric computation. *Computational Geometry*, 7(1-2):3–23, 1997.