Computable decision making on the reals and other spaces

via partiality and nondeterminism

Benjamin Sherman
MIT CSAIL
sherman@csail.mit.edu

Luke Sciarappa
MIT CSAIL
lukesci@mit.edu

Adam Chlipala
MIT CSAIL
adamc@csail.mit.edu

Michael Carbin
MIT CSAIL
mcarbin@csail.mit.edu

Abstract

Though many safety-critical software systems use floating point to represent real-world input and output, the mathematical specifications of these systems' behaviors use real numbers. Significant deviations from those specifications can cause errors and jeopardize safety. To ensure system safety, some programming systems offer exact real arithmetic, which often enables a program's computation to match its mathematical specification exactly. However, exact real arithmetic complicates decision-making: in these systems, it is impossible to compute (total and deterministic) discrete decisions based on connected spaces such as $\mathbb{R}$. We present programming-language semantics based on constructive topology with variants allowing nondeterminism and/or partiality. Either nondeterminism or partiality suffices to allow computable decision making on connected spaces such as $\mathbb{R}$. We then introduce *pattern matching on spaces*, a language construct for creating programs on spaces, generalizing pattern matching in functional programming, where patterns need not represent decidable predicates and also may overlap or be inexhaustive, giving rise to nondeterminism or partiality, respectively. Nondeterminism and/or partiality also yield formal *logics for constructing approximate decision procedures*. We extended the Marshall language for exact real arithmetic with these constructs and implemented some programs with it.

CCS Concepts • **Theory of computation → Constructive mathematics**; Program semantics; • **Mathematics of computing →** *Continuous functions*; Point-set topology;

Keywords locale theory, constructive analysis

ACM Reference Format:
Benjamin Sherman, Luke Sciarappa, Adam Chlipala, and Michael Carbin. 2018. Computable decision making on the reals and other spaces: via partiality and nondeterminism. In *Proceedings of 33rd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '18).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3209108.3209193

1 Introduction

Ensuring the safety of software that mixes discrete and continuous computation—such as cyber-physical systems, numerical computations, and machine learning—can be challenging. When continuous values are represented unsoundly, such as with finite precision, failure can result from numerical error alone. Verification necessitates both guaranteeing accuracy of computations with continuous values and idealized reasoning about a system's behavior.

LICS '18, July 9–12, 2018, Oxford, United Kingdom

ACM ISBN 978-1-4503-5583-4/18/07.
https://doi.org/10.1145/3209108.3209193

Programming systems that implement exact real arithmetic [Bauer 2008; O'Connor 2008; Taylor 2010] do guarantee accuracy and have been used to develop verified cyber-physical systems [Anand and Knepper 2015]. While these programming systems ease development of traditionally continuous computations on the reals, there has been little investigation of how to soundly incorporate decision-making: computations from the reals ($\mathbb{R}$) to the Booleans ($\mathbb{B}$). Classic results prove that it is impossible to compute (total and deterministic) discrete decisions based on connected spaces such as $\mathbb{R}$ [Weihrauch 1995].

However, we show that by allowing partiality or nondeterminism into the computational model, we can enable decision-making while retaining fairly strong computational abilities. We present programming-language semantics based on *constructive topology* with variants allowing nondeterminism and/or partiality.

Constructive topology, in the form of *locale theory*, provides a single programming language in which it is possible to build and execute programs that compute with continuous values and to reason about these programs in terms of their mathematical descriptions. In this programming language (category) **FSpc**, types (objects) are *spaces* and programs (morphisms) are *continuous maps*.

Types are spaces. Spaces are defined as theories of *geometric logic* [Vickers 2007a]: propositional symbols describe the core observable properties of the space, and axioms describe which properties imply others. Points of a space are models of its theory.

For example, the theory for $\mathbb{R}$ has as its propositional symbols the open balls with rational centers, $q - \varepsilon < \cdot < q + \varepsilon$ (for each $q : \mathbb{Q}$, $\varepsilon : \mathbb{Q}^+$), and an example axiom is the one $\top \leq \bigvee_{q:\mathbb{Q}} (q - \varepsilon < \cdot < q + \varepsilon)$ that says (for any $\varepsilon : \mathbb{Q}^+$) that without assumptions ($\top$), a point must be within ε of *some* rational q. Axioms with disjunctions on the right, like this one, are called *open covers*. They can be read computationally. For instance, since the point π lies in $\top$ (as every point does), it must lie in some ball of radius 1 with a rational center, and indeed it should be able to *compute* any such rational: 3 would be one possible choice, since $3 - 1 < \pi < 3 + 1$.

Programs are continuous maps. One defines a continuous map $f : A \to_c B$ by programming how it reduces open covers of B to open covers of A. Accordingly, computation is pull-based, where a composition of functions successively reduces open covers of the output to open covers of the input, at which point the input computes which open of the cover it lies in, which corresponds to a particular open that the output lies in. Constructive topology has surprisingly strong computational abilities [Escardó 2007; Simpson 1998; Taylor 2010], such as the ability to compute the maximum that a real-valued function attains over a compact-overt space (see Definition 5.4).

1.1 Contributions

Making nontrivial total and deterministic decisions based on connected spaces is impossible: any continuous map $f : C \to_c D$ from

a connected space C to a discrete space D must be constant. We demonstrate that decisions *can* be made, however, by permitting either partiality or nondeterminism, and we continue to then present the following contributions:

Partial and/or nondeterministic maps. §3 defines partiality and nondeterminism as they relate to continuous maps and §4.2 characterizes the open maps and open embeddings as those continuous maps having partial and/or nondeterministic inverses. While each of these subjects has been studied individually in the context of constructive topology, we contribute the first integrated characterization relating them.

Pattern matching on spaces. §4 generalizes pattern matching on inductive types in functional programming to spaces. It differs in that patterns need not correspond to decidable predicates, and patterns are allowed to overlap or fail to be injective, yielding nondeterminism, or be inexhaustive, yielding partiality.

Formal logics for approximate decision procedures. §5 generalizes the decidable predicates of functional programming to *approximately* decidable predicates on spaces, which may either be partial or nondeterministic. Because spaces often have few decidable predicates, this relaxation is essential for decision-making on spaces. The Boolean algebra of decidable predicates generalizes to a quasi-Boolean algebra, and quantification over finite sets is generalized to quantification over *compact-overt* spaces (see Definition 5.4). The partial logic and the nondeterministic logic are observed to be duals of each other.

Case study. We have extended the Marshall language [Bauer 2008] for exact real arithmetic with versions of these constructs (§6) and in §7 present two example programs that make critical use of those constructs to solve decision-making tasks.

Our results show how constructive topology can serve as the foundation for programming systems that support mixing discrete and continuous computation. The hope is that such programming systems can dramatically simplify the development and verification of applications—such as cyber-physical systems, numerical computations, and machine learning—that must make decisions based on continuous values.

A longer version of this paper [Sherman et al. 2018] provides further technical detail and proofs.

2 Constructive topology

This section reviews locale theory, a constructive theory of topology that provides a semantic and computational foundation for programming with spaces. Readers interested in a more thorough introduction may wish to consult Vickers's *Topology via Logic* 1989.

Preliminaries. We intend mathematical statements to be interpreted within a constructive metatheory with a universe of impredicative propositions Ω, potentially formalizable within, for instance, the Calculus of Constructions[1]. We use the term "type" to refer to a type and the term "set" to refer to what is often called a setoid or a Bishop set [Bishop 1967]: a type together with a distinguished equivalence relation on it, which we denote by $=$. If A and B are both sets, then the notation $f : A \to B$ means that f is a morphism of sets (i.e., it maps equivalent elements of A to equivalent elements of B). For objects A, B of a category, let the notation $A \cong B$ indicate that they are isomorphic.

Definition 2.1. A *space*[2] A is a distributive lattice $\mathcal{O}(A)$ that has top and bottom elements, $\top$ and $\bot$, respectively, and that has all joins such that binary meets distribute over all joins:

$$U \wedge \bigvee_{i:I} V_i = \bigvee_{i:I} U \wedge V_i.$$

We call the lattice $\mathcal{O}(A)$ the *opens* of A. This lattice describes the observable or "affirmable" properties of A [Vickers 1989]. If $U \le \bigvee_{i:I} V_i$, we call the family $(V_i)_{i:I}$ an *open cover* of U.

Definition 2.2. A *point* x of a space A is a subset $(x \models \cdot) : \mathcal{O}(A) \to \Omega$ (read "x lies in") such that

$$\frac{x \models U \qquad U \le \bigvee_{i:I} V_i}{\exists i : I.\ x \models V_i}\ \textsc{join} \qquad \frac{}{x \models \top}\ \textsc{meet-0}$$

$$\frac{x \models U \qquad x \models V}{x \models U \wedge V}\ \textsc{meet-2}.$$

The formal proof that a point satisfies the above three rules both justifies the consistency of its definition and provides its computational content. Intuitively, $x \models U$ means we have some knowledge U about x. JOIN says that it is possible to refine existing knowledge about x to get an even sharper estimate of where x lies. When a point x that lies in U is presented with an open cover $U \le \bigvee_{i:I} V_i$, it uses the *proof* of the covering relationship to *compute* some open V_i that x also lies in. The index i is a concrete answer that indicates where the point lies. MEET-0 says that we know *something* about x (which we can then refine with JOIN), and MEET-2 says that we can assimilate two pieces of knowledge about x into one, which assures that they are mutually consistent.

Definition 2.3. A *continuous map* $f : A \to_c B$ between spaces is a map $f^* : \mathcal{O}(B) \to \mathcal{O}(A)$, called an *inverse image map*, that preserves all joins, $\top$, and binary meets, i.e., it satisfies

$$\frac{U \le \bigvee_{i:I} V_i}{f^*(U) \le \bigvee_{i:I} f^*(V_i)}\ \textsc{join} \qquad \frac{}{\top \le f^*(\top)}\ \textsc{meet-0}$$

$$\frac{}{f^*(U) \wedge f^*(V) \le f^*(U \wedge V)}\ \textsc{meet-2}.$$

A continuous map $f : A \to_c B$ transforms covers on B into covers on A. Spaces and continuous maps form a cartesian monoidal category we call **FSpc** (for *formal spaces*)[3]. The terminal object is the one-point space $*$, whose lattice of opens $\mathcal{O}(*)$ is Ω, where $U \le V$ if U implies V. Points of a space A can be identified with continuous maps $* \to_c A$, and in particular the JOIN and MEET rules for continuous maps reduce to the corresponding rules for

[1] It is possible to formulate a predicative analogue of locale theory known as *formal topology*, which makes more clear the computational content of constructive topology. This does impose some difficulties that require some changes. For instance, the construction of product spaces is generally impredicative, but it is possible to instead use *inductively generated formal spaces* [Coquand et al. 2003], which has products even in a predicative setting. All spaces used in this paper are inductively generated [Coquand et al. 2003; Vickers 2004, 2005, 2007b, 2009]. Palmgren 2003 offers a more careful treatment of predicativity and universes in formal topology.

[2] Since all topological notions in this article are pointfree, we coopt terminology from classical topology without fear of confusion. For instance, we say "space" rather than "locale" when describing the pointfree analogue of spaces.

[3] **FSpc** is often called **Loc**, for *locales*.

points. Two continuous maps are equal if they have the same inverse image maps. One can think of the inverse image map as a behavioral *specification* and the formal proof that the continuous map preserves meets and finitary joins as an *implementation* of that specification.

Given a space A and an open $U : \mathcal{O}(A)$, we can form the open subspace $\{A \mid U\}$ of A by making $\mathcal{O}(\{A \mid U\})$ a quotient of $\mathcal{O}(A)$, identifying opens $P, Q : \mathcal{O}(A)$ in $\{A \mid U\}$ when $P \wedge U = Q \wedge U$.

3 Decision making with partiality and nondeterminism

The real line $\mathbb{R}$ is *connected*, meaning that any continuous map $f : \mathbb{R} \to_c D$ to a discrete set D must be a constant map. In particular, every map $f : \mathbb{R} \to_c \mathbb{B}$ is constant. The practical implications of connectedness are severe: it is impossible to (continuously) make (nontrivial) discrete decisions over variables that come from connected spaces such as $\mathbb{R}$.

Proposition 3.1. *Continuous maps $f : A \to_c \mathbb{B}$ are in bijective correspondence with pairs of opens (P, Q) of A that are* covering*, i.e., $\top \leq P \vee Q$, and* disjoint*, i.e., $P \wedge Q \leq \bot$.*

Proof sketch. Since f^* preserves joins, it is specified entirely by its behavior on the two basic opens, $P \triangleq f^*(\cdot = \mathsf{true})$ and $Q \triangleq f^*(\cdot = \mathsf{false})$. Since f^* preserves $\top$ (MEET-0), P and Q are covering, and since f^* preserves binary meets (MEET-2), P and Q are disjoint. □

While it is impossible to make discrete decisions on connected spaces A that are total and deterministic, we *can* make decisions that are either partial (only defined on some open subspace of the input space) or nondeterministic (could potentially give different answers even when given the exact same input). Partiality relaxes the requirement that the inverse image map preserves $\top$ (MEET-0), while nondeterminism relaxes the requirement that the inverse image map preserves binary meets (MEET-2). Accordingly, partial $\mathbb{B}$-valued maps correspond to pairs of opens that are not necessarily covering, and nondeterministic $\mathbb{B}$-valued maps correspond to pairs of opens that are not necessarily disjoint.

In this section, we present categories whose objects are spaces and whose morphisms are like continuous maps, but the inverse image maps need not necessarily preserve $\top$ or binary meets (see Fig. 1). The remainder of this section characterizes these partial and/or nondeterministic maps and the monads that represent them.

3.1 Partiality

The MEET-0 rule enforces totality: viewing $\top$ as the predicate representing the entire space, MEET-0 says that a point must lie in the entire space. Eliminating MEET-0 permits definition of a continuous map that is only defined on an open subspace of the domain.

Definition 3.2. A *partial map* f from A to B, written $f : A \to_p B$, is a map $f^* : \mathcal{O}(B) \to \mathcal{O}(A)$ that preserves joins and binary meets but not necessarily $\top$. These maps form a category $\mathbf{FSpc}_p$.

Example 3.3. Consider the task of comparing a real number with 0. We can define a partial comparison $\mathsf{cmp} : \mathbb{R} \to_p \mathbb{B}$ by only defining a continuous map on the open subspace $\{\mathbb{R} \mid \cdot \neq 0\}$ of $\mathbb{R}$. We specify its observable behavior with its inverse image map

$$\mathsf{cmp}^*(\cdot = \mathsf{true}) \triangleq \cdot > 0 \qquad \mathsf{cmp}^*(\cdot = \mathsf{false}) \triangleq \cdot < 0.$$

The inverse image map cmp^* in fact defines a partial map, as cmp^* preserves joins and binary meets, but it is not total, since it fails to preserve $\top$.

Proof. To confirm that cmp^* preserves binary meets, it suffices to check binary meets of distinct basic opens, so we confirm

$$\begin{aligned}\mathsf{cmp}^*((\cdot = \mathsf{true}) \wedge (\cdot = \mathsf{false})) = \mathsf{cmp}^*(\bot) = \bot = (\cdot > 0) \wedge (\cdot < 0)\\ = \mathsf{cmp}^*(\cdot = \mathsf{true}) \wedge \mathsf{cmp}^*(\cdot = \mathsf{false}).\end{aligned}$$

However, cmp^* does not preserve $\top$, since $\top \not\leq \mathsf{cmp}^*(\top) = (\cdot < 0) \vee (\cdot > 0)$. □

There is a bijective correspondence between partial maps and continuous maps defined on some open subspace of the domain.

3.2 Nondeterminism

The MEET-2 rule enforces determinism. Spatially, the rule says that if a point lies in two opens, it must lie in their intersection. Computationally, it says that it should be possible to consistently reconcile different answers given by different refinements computed by use of the JOIN rule. Eliminating MEET-2 allows the definition of programs whose observable behavior might depend on the exact implementation of their inputs (specifically, the formal proofs that their inputs preserve joins and finitary meets). Rather than viewing such behavior as breaking the abstraction provided by the equivalence relation on points (since points that lie in the same opens may be treated differently), we can instead choose to maintain this abstraction and view such behavior as fundamentally nondeterministic.

Definition 3.4. A *nondeterministic map* f from A to B, written $f : A \to_{nd} B$, is a map $f^* : \mathcal{O}(B) \to \mathcal{O}(A)$ that preserves joins and $\top$ but not necessarily binary meets. These maps form a category $\mathbf{FSpc}_{nd}$.

For instance, we can perform a nondeterministic approximate comparison of a real number with 0:

Example 3.5. Fix some error tolerance parameter $\varepsilon > 0$. We may define a total but nondeterministic approximate comparison with 0, $\mathsf{cmp} : \mathbb{R} \to_{nd} \mathbb{B}$, allowing error up to ε, by specifying its observable behavior with the inverse image map

$$\mathsf{cmp}^*(\cdot = \mathsf{true}) \triangleq \cdot > -\varepsilon \qquad \mathsf{cmp}^*(\cdot = \mathsf{false}) \triangleq \cdot < \varepsilon.$$

We can confirm that cmp^* in fact defines a nondeterministic map, as it preserves joins and $\top$ but fails to preserve binary meets.

Proof. Since cmp's codomain is discrete, it trivially satisfies JOIN. We confirm it preserves $\top$:

$$\begin{aligned}\mathsf{cmp}^*(\top) &= \mathsf{cmp}^*(\cdot = \mathsf{true}) \vee \mathsf{cmp}^*(\cdot = \mathsf{false})\\ &= (\cdot > -\varepsilon) \vee (\cdot < \varepsilon) = \top.\end{aligned}$$

However, it fails to preserve binary meets, since

$$\mathsf{cmp}^*((\cdot = \mathsf{true}) \wedge (\cdot = \mathsf{false})) = \mathsf{cmp}^*(\bot) = \bot$$

but

$$\mathsf{cmp}^*(\cdot = \mathsf{true}) \wedge \mathsf{cmp}^*(\cdot = \mathsf{false}) = (\cdot > -\varepsilon) \wedge (\cdot < \varepsilon) = -\varepsilon < \cdot < \varepsilon,$$

which is not $\bot$. □

3.3 Both partiality and nondeterminism

Definition 3.6. A *nondeterministic and partial map* f from A to B, written $f : A \to_{nd,p} B$, is a map $f^* : \mathcal{O}(B) \to \mathcal{O}(A)$ that preserves joins but not necessarily any meets. These maps form a category $\mathbf{FSpc}_{nd,p}$ (equivalent to the category of suplattices).

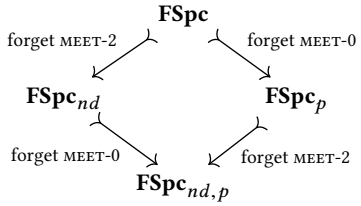

Figure 1. The lattice of categories representing potentially nondeterministic (*nd*) or partial (*p*) maps on spaces.

3.4 Monads and summary

Potentially allowing partiality and nondeterminism yields a lattice of categories that represent nondeterministic and partial maps, depicted in Fig. 1, where each arrow denotes a faithful ("forgetful") functor where a particular rule is no longer required for inverse image maps. These forgetful functors have right adjoints, such that they induce a family of (strong) monads on **FSpc**: $\cdot_\perp$ for representing partiality, $\mathcal{P}^+_\Diamond$ for nondeterminism, and $\mathcal{P}_\Diamond$ for both[4]. Their adjunctions give the correspondences

$$\begin{aligned} A \to_{nd,p} B &\cong A \to_c \mathcal{P}_\Diamond(B) \\ A \to_{nd} B &\cong A \to_c \mathcal{P}^+_\Diamond(B) \\ A \to_p B &\cong A \to_c B_\perp. \end{aligned}$$

Accordingly, it is possible to use these monads to have access to partiality and/or nondeterminism within the language of continuous maps.

4 Pattern matching

Often, programmers would like to compose a decision on $\mathbb{R}$ with other computations that depend on the decision and thus associate each condition with a corresponding computation. This programming pattern resembles pattern matching in traditional functional programming, and therefore, in this section, we identify and present general pattern matching for spaces. Our constructions admit partial and/or nondeterministic pattern matches, where in the former case the collection of patterns may not be exhaustive, and in the latter they may overlap. While the syntax of a pattern match determines a unique map that is potentially partial and nondeterministic, there are simple conditions that ensure that a map is total or deterministic:

1. *Totality*: together, the cases cover the entire input space.
2. *Determinism*: patterns are disjoint and injective.

4.1 Pattern families

This section characterizes those families of patterns that may be used to match on a scrutinee that comes from a space A (if nothing is to be assumed about the branches). The idea is that we compose a function $f : A \to_C B$ by factoring through a disjoint sum over a collection of spaces representing the possible patterns and branches, $\sum_{i:I} U_i$, i.e., a composition

$$A \xrightarrow{\mathsf{inv}} \textstyle\sum_{i:I} U_i \xrightarrow{e} B$$

of a "pattern matching" part inv followed by the "branch execution" part e. The collection of branches $(e_i : U_i \to_C B)_{i:I}$ exactly correspond to the branch-execution function e, but the pattern-matching part inv is more interesting; this section will address those families of patterns that may yield valid functions of this sort.

Semantically, we think of a single pattern as representing a space U together with a map $p : U \to_c A$ that represents the possibility that the scrutinee can be represented as a point in the image of p. For a single pattern $p : U \to_c A$ to be implementable, it must have a well-behaved inverse $p^{-1} : A \to_{nd,p} U$ that is partial and also may be nondeterministic. If we are building a program that is total, then we do not need each p^{-1} to be total, but we do need the collection of them to cover A. If we are building a program that is deterministic, then p^{-1} should be deterministic. We will find that *open maps* are exactly those with well-behaved (nondeterministic and partial) inverses, and *open embeddings* are open maps whose inverses are deterministic.

4.2 Open maps and open embeddings

In pattern matching for functional programming, one may pattern match on an inductive type by checking whether it has the form of a particular constructor applied to some argument (i.e., it is in the image of the map defined by a particular constructor). The analogues of constructors for pattern matching on spaces are the open maps and the open embeddings.

4.2.1 Open maps

Definition 4.1 (Johnstone 2002)**.** A continuous map $f : A \to_c B$ is an *open map* (which we may denote by $f : A \to_o B$) if the inverse image map $f^* : \mathcal{O}(B) \to \mathcal{O}(A)$ has a left adjoint $f_! : \mathcal{O}(A) \to \mathcal{O}(B)$, called the *direct image map*, that satisfies the Frobenius law, $f_!(U \wedge f^*(V)) = f_!(U) \wedge V$.

That is, f is an open map if the *image* of any open in A is open in B; the direct image map provides this mapping.

Proposition 4.2. *For any open map $p : A \to_c B$, there is a (potentially) partial and nondeterministic inverse map $p^{-1} : B \to_{nd,p} A$ whose inverse image map is $p_!$.*

Proof. We only must prove that $p_!$ preserves joins: it does, since $p_!$ is a left adjoint (to p^*). □

An example of an open map is the "return" function of the nondeterminism monad $\{\cdot\} : A \to_c \mathcal{P}^+_\Diamond(A)$.

4.2.2 Open embeddings

Given an open U of a space A, let $\iota[U] : \{A \mid U\} \to_c A$ denote the inclusion of the open subspace $\{A \mid U\}$ into A.

Lemma 4.3. *An open map $f : A \to_c B$ factors through its direct image $f_!(\top)$, i.e., there is an $\tilde{f}$ such that the following diagram commutes:*

$$\begin{array}{ccc} A & \xrightarrow{\tilde{f}} & \{B \mid f_!(\top)\} \\ & \searrow^{f} & \downarrow{\scriptstyle \iota[f_!(\top)]} \\ & & B \end{array}$$

Definition 4.4. A map $f : A \to_c B$ is an *open embedding* (or *open inclusion*), denoted $f : A \hookrightarrow B$, if A is isomorphic to its image under f in B, i.e., if there is an open $U : \mathcal{O}(B)$ and isomorphism $\tilde{f} : A \to_c \{B \mid U\}$ such that the following diagram commutes:

$$\begin{array}{ccc} A & \underset{\tilde{f}^{-1}}{\overset{\tilde{f}}{\rightleftarrows}} & \{B \mid U\} \\ & \searrow^{f} & \downarrow{\scriptstyle \iota[U]} \\ & & B \end{array}$$

[4] Each of these strong monads preserves inductive generation of spaces [Vickers 1989, 2004].

Theorem 4.5. *A map $f : A \to_c B$ is an open embedding if and only if it is an open map and its direct image map $f_!$ preserves binary meets.*

Proof sketch. Given an open embedding $f : A \to_c B$ that factors through $\{B \mid U\}$, letting $\tilde{f}$ and $\tilde{f}^{-1}$ be the maps as in the above diagram, its direct image map is given by

$$f_! : \mathcal{O}(A) \to \mathcal{O}(B)$$
$$f_!(V) \triangleq \tilde{f}^{-1*}(V) \wedge U.$$

Conversely, given an open map $f : A \to_c B$ with a meet-preserving direct image map $f_!$, we claim that $A \cong \{B \mid f_!(\top)\}$. □

Proposition 4.6. *Given an open embedding $f : A \hookrightarrow B$, the "inverse" map $f^{-1} : B \to_{nd,p} A$ that any open map has is in fact deterministic, i.e., $f^{-1} : B \to_p A$.*

Proof. Follows directly from the fact that $f_!$ preserves joins and binary meets. □

An example of an open embedding is the "return" function of the partiality monad $\mathsf{up} : A \to_c A_\perp$. This allows us to view A as an open subspace of $A_\perp$.

4.3 Pattern families: definition and properties

In general, we have an entire family of patterns $(p_i : U_i \to_o A)_{i:I}$ where I is some index type. We can use this pattern family to construct the partial and nondeterministic inverse

$$\mathsf{inv} : A \to_{nd,p} \sum_{i:I} U_i$$
$$\mathsf{inv}(x) \triangleq \bigsqcup_{i:I} \mathsf{inj}_i(p_i^{-1}(x)),$$

where $\sqcup$ denotes the nondeterministic join, $\sqcup : \prod_{i:I} X \to_{nd,p} X$ for any space X.

Therefore, for any index type I, any collection of open maps $p_i : U_i \to_o A$ is a collection of patterns for defining a nondeterministic and partial map using a pattern match. Given an arbitrary collection of branches $e_i : U_i \to_{nd,p} B$, or equivalently, $e : \sum_{i:I} U_i \to_{nd,p} B$, the pattern match is just the composition $e \circ \mathsf{inv} : A \to_{nd,p} B$. As expected, the pattern match is a nondeterministic union of its branches.

For those categories/languages that require totality or determinism, we would like to characterize the families of patterns that are suitable in those cases. These will be subcollections of the collection of pattern families in the nondeterministic and partial case.

Definition 4.7. A *pattern family* for a subcategory C of $\mathbf{FSpc}_{nd,p}$ is a family of open maps $(p_i : U_i \to_o A)_{i:I}$ such that $\bigsqcup_{i:I} \mathsf{inj}_i \circ p_i^{-1} : A \to_{nd,p} \sum_{i:I} U_i$ is in C. Let $\mathcal{J}_C$ denote the collection of pattern families for C.

Theorem 4.8. *We can characterize the pattern families for the following subcategories of $\mathbf{FSpc}_{nd,p}$ as exactly those families of open maps $(p_i : U_i \to_o A)_{i:I}$ satisfying certain additional properties:*

- $\mathbf{FSpc}_{nd}$ *(*Totality*) The patterns cover the whole input space, i.e.,* $\top \le \bigvee_{i:I} p_{i!}(\top)$.
- $\mathbf{FSpc}_{p}$ *(*Determinism*) Each $p_i : U_i \to_o A$ is an open embedding, and the patterns are* pairwise disjoint*, meaning that whenever $p_{i!}(\top) \wedge p_{j!}(\top)$ is positive*[5] *in A, then (intensionally) $i = j$.*
- $\mathbf{FSpc}$ *(*Totality *and* determinism*) The above conditions for totality and determinism must both hold.*

[5] An open U is called *positive* if whenever $U \le \bigvee_{i:I} V_i$, I is inhabited, i.e., every open cover of U itself must have at least one open.

For any subcategory C of $\mathbf{FSpc}_{nd,p}$, one can construct a pattern match by composing a pattern family $(p_i : U_i \to_o A)_{i:I}$ for C with a collection of branches $e : \sum_{i:I} U_i \to_C B$ that is in C.

When determinism is required, we recover the familiar condition required of pattern matching in functional programming: disjointness of patterns (i.e., patterns are not allowed to overlap). When both determinism and totality are required, we further recover the familiar condition that pattern membership is decidable:

Proposition 4.9. *For a pattern family $(p_i : U_i \hookrightarrow A)_{i:I}$ for **FSpc**, if the index type I has decidable equality, then for each $i : I$, there is a map $\chi_{p_{i!}(\top)} : U_i \to_c \mathbb{B}$ satisfying $\chi^*_{p_{i!}(\top)}(\cdot = \mathsf{true}) = p_{i!}(\top)$ (i.e., $p_{i!}(\top)$ is clopen).*

We will now observe that the pattern families for the various subcategories of $\mathbf{FSpc}_{nd,p}$ form a lattice of Grothendieck pretopologies. This tells us that there are certain techniques that we can always use to form pattern families, and that pattern families will have important structural properties. For instance, the transitivity axiom corresponds to the ability to flatten nested pattern matches into a single one. The stability axiom allows us to use "pulled-back covers": it is possible to pattern match on an input $x : A$ by doing a case analysis on $f(x) : B$, such that in each branch it is known that x lies in a particular open subspace of A (rather than only knowing that $f(x)$ lies in a particular open subspace of B). The root-finding example in §7.2 uses a pulled-back cover in this way.

Definition 4.10 (Mac Lane and Moerdijk 1992)**.** A *Grothendieck pretopology* is an assignment to each space A a collection of families $(U_i \to_c A)_{i:I}$ of continuous maps, called *covering families*, such that

1. *isomorphisms cover* – every family consisting of a single isomorphism $U \xrightarrow{\cong}_c A$ is a covering family;
2. *stability axiom* – the collection of covering families is stable under pullback: if $(U_i \to_c A)_{i:I}$ is a covering family and $f : V \to_c A$ is any continuous map, then the family of pullbacks $(f^* U_i \to_c V)_{i:I}$ is a covering family;
3. *transitivity axiom* – if $(U_i \to_c A)_{i:I}$ is a covering family and for each i also $(U_{i,j} \to_c U_i)_{j:J_i}$ is a covering family, then also the family of composites $(U_{i,j} \to_c U_i \to_c A)_{i:I, j:J_i}$ is a covering family.

Proposition 4.11 (Product axiom)**.** *In any Grothendieck pretopology, given covering families $(p_i : U_i \to_c A)_{i:I}$ and $(q_j : V_j \to_c B)_{j:J}$, there is a product covering family $(p_i \otimes q_j : U_i \times V_j \to_c A \times B)_{(i,j):I\times J}$.*

Theorem 4.12. *For each $C \in \{\mathbf{FSpc}, \mathbf{FSpc}_{nd}, \mathbf{FSpc}_{p}, \mathbf{FSpc}_{nd,p}\}$, the collection of pattern families for C, $\mathcal{J}_C$, forms a Grothendieck pretopology.*

4.4 Syntax of pattern matching

We now describe syntax for a programming language with pattern matching as guided by the above semantics (see Fig. 2).

Each of $C \in \{\mathbf{FSpc}, \mathbf{FSpc}_{nd}, \mathbf{FSpc}_{p}, \mathbf{FSpc}_{nd,p}\}$ are cartesian monoidal categories, meaning that they admit a restricted (first-order) λ-calculus syntax [Escardó 2004], with the typing rules

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash_C x : A}\ \textsc{var} \qquad \frac{f : A_1 \times \cdots \times A_n \to_C B \qquad \Gamma \vdash_C e_1 : A_1 \quad \cdots \quad \Gamma \vdash_C e_n : A_n}{\Gamma \vdash_C f(e_1, \ldots, e_n) : B}\ \textsc{app},$$

$$\text{expression } e ::= x \mid f(e, \ldots, e) \mid \text{case}(e)\begin{cases} p \Rightarrow e \\ \vdots \\ p \Rightarrow e \end{cases}$$

$$\text{pattern } p ::= f \mid x \mid \_ \mid f(p) \mid p, p$$

function f

variable x

Figure 2. Syntax for our simple language with pattern matching.

where contexts are lists of spaces and in the VAR rule $(x : A) \in \Gamma$ denotes a witness that A is a member of the list Γ. Let $\text{Prod} : \text{list}(\mathbf{FSpc}) \to \mathbf{FSpc}$ denote the product of a list of types. We use an unadorned turnstile $\vdash$ for continuous maps, $C = \mathbf{FSpc}$.

Proposition 4.13. *Given any expression $\Gamma \vdash_C e : A$, we can construct a term $[\![e]\!] : \text{Prod}(\Gamma) \to_C A$.*

In Fig. 3 we define typing rules for patterns, where $p : A \dashv \Gamma$ intuitively means that the pattern p provides a context of pattern matching variables Γ by pattern matching on a space A. For instance, we could have the pattern

$$(\text{up}(x), y) : A_\perp \times B \dashv x : A, y : B$$

on a space $A_\perp \times B$ that provides variables $x : A$ and $y : B$ to be used in the branch corresponding to that pattern.

Theorem 4.14. *Given any pattern derivation $p : A \dashv \Gamma$, there is a map $[\![p]\!] : \text{Prod}(\Gamma) \times \Delta \to_o A$ for some space Δ that collects the "discarded variables" from the wildcards*[6].

Proof sketch. Follows from the fact that open maps include the identity (VAR, WILDCARD) and are closed under composition (COMPOSE) and parallel composition (PRODUCT). □

Note that the same pattern syntax would work with open embeddings instead of open maps, as they too form a category and are closed under parallel composition; patterns of open embeddings are necessary for constructing deterministic programs.

We can now define a general (mostly) syntactic rule for interpreting the pattern matches described in this section.

Theorem 4.15. *We can interpret the syntax*[7]

$$\frac{\Gamma \vdash_C s : A \qquad \prod_{i:I} p_i : A \dashv_o A_i \qquad \prod_{i:I} \Gamma, A_i \vdash_C e_i : B \qquad ([\![p_i]\!])_{i:I} \in \mathcal{J}_C}{\Gamma \vdash_C \left(\text{case}(s)\left\{[i : I] \quad p_i \quad \Rightarrow e_i\right\}\right) : B}\ \textit{CASE-C}$$

(with the one non-syntactic side condition $([\![p_i]\!])_{i:I} \in \mathcal{J}_C$), where $C \in \{\mathbf{FSpc}, \mathbf{FSpc}_{nd}, \mathbf{FSpc}_p, \mathbf{FSpc}_{nd,p}\}$.

Proof. The syntactic constructions give us maps $[\![s]\!] : \Gamma \to_C A$, $[\![p_i]\!] : A_i \times \Delta_i \to_o A$ (for some spaces Δ_i representing discarded variables in the pattern p_i), and $[\![e_i]\!] : \Gamma \times A_i \to_C B$. The condition $([\![p_i]\!])_{i:I} \in \mathcal{J}_C$ means that these maps appropriately cover A, so that we get an appropriately behaved map $\text{inv} : A \to_C \sum_{i:I} A_i \times \Delta_i$.

[6] Discarding variables is unnecessary if we only consider overt spaces A; a space A is overt if the map $A \to_c *$ is an open map. Classically, all spaces are overt.

[7] As mnemonics, I stands for *index* type, s stands for *scrutinee* of a case expression, p stands for *pattern*, and e for *expression*.

$$\frac{f : * \to_o A}{f : A \dashv \cdot}\ \text{CONSTANT} \qquad \frac{}{v : A \dashv v : A}\ \text{VAR}$$

$$\frac{}{\_ : A \dashv \cdot}\ \text{WILDCARD} \qquad \frac{p : U \dashv \Gamma \qquad f : U \to_o A}{f(p) : A \dashv \Gamma}\ \text{COMPOSE}$$

$$\frac{p : A \dashv \Gamma \qquad q : B \dashv \Delta}{p, q : A \times B \dashv \Gamma, \Delta}\ \text{PRODUCT}$$

Figure 3. Typing rules for patterns.

We must produce a map $f : \Gamma \to_C B$. We can do so by defining[8]

$$f : \Gamma \to_C B$$
$$f(\gamma) \triangleq \text{let } \langle i, x\rangle \triangleq \text{inv}([\![s]\!](\gamma)) \text{ in } [\![e_i]\!](\gamma, \text{fst}(x)). \qquad \square$$

Note that the condition $([\![p_i]\!])_{i:I} \in \mathcal{J}_C$ that the patterns lie in the appropriate Grothendieck pretopology is trivial when $C = \mathbf{FSpc}_{nd,p}$, making the rule purely syntactic. For $C \neq \mathbf{FSpc}_{nd,p}$, it would be interesting to study when the non-syntactic covering and/or disjointness requirements can be decided automatically.

5 Formal logics for approximate decision procedures

In this section, we develop a formal logic for constructing decision procedures on spaces that may be either partial or nondeterministic, by simply considering partial and/or nondeterministic Boolean values. This is useful where there are not total and deterministic decision procedures: for instance, there are no nontrivial maps $\mathbb{R} \to_c \mathbb{B}$ (since $\mathbb{R}$ is connected) but plenty of nondeterministic maps $\mathbb{R} \to_{nd} \mathbb{B}$ or partial maps $\mathbb{R} \to_p \mathbb{B}$.

Conventional decision procedures in general functional programming correspond to decidable predicates, or functions returning Boolean values. Decidable predicates are closed under conjunction, disjunction, and negation since those operations are computable on $\mathbb{B}$, and additionally they are closed under universal and existential quantification over finite[9] sets.

Analogously, $\mathbb{B}$-valued continuous maps are also closed under conjunction, disjunction, and negation, and they admit quantification over *compact-overt* spaces (which generalize finite sets). We will show that these operations still work when partiality or nondeterminism is admitted, where Boolean logic is generalized to many-valued logic with the structure of a quasi-Boolean algebra.

Proposition 5.1. *Maps $A \to_{nd,p} \mathbb{B}$ are in bijective correspondence with pairs (P, Q) of opens of A. For maps $A \to_{nd} \mathbb{B}$ the opens are* covering, *i.e., $\top \leq P \vee Q$, and for maps $A \to_p \mathbb{B}$ they are* disjoint, *i.e., $P \wedge Q \leq \bot$.*

This correspondence between pairs of opens and Boolean-valued maps establishes two different perspectives on approximate decision-making, one spatial and one more algorithmic in flavor. We think of P as the "true" region and Q the "false" region. We can also think of Q as representing the closed subspace complement $\overline{Q}$ of the open

[8] While the "let-in" syntax for using the universal property (itself a sort of pattern matching) of sums has not been formally described, hopefully it is clear how it can be implemented via categorical semantics.

[9] By "finite," we always mean Kuratowski-finite [Johnstone 1977].

subspace Q, in which case $T \le P \vee Q$ corresponds to the subspace inclusion $\overline{Q} \subseteq P$, and $P \wedge Q \le \bot$ corresponds to $P \subseteq \overline{Q}$.

Adding nondeterminism or partiality changes the behavior in comparison to deterministic decision procedures. We can characterize this algebraically. In **Set**, $\mathbb{B}$ forms a Boolean algebra. Since the functor Discrete : **Set** $\to$ **FSpc** is full, faithful, and preserves binary products and the terminal object, this lifts to an internal Boolean algebra (within **FSpc**) on $\mathbb{B}$ the space. We can again lift these operations by the forgetful functor **FSpc** $\to$ $\mathbf{FSpc}_C$; for instance, the lifted version of the Boolean "and" operation ($\&\& : \mathbb{B}\times\mathbb{B} \to_{nd,p} \mathbb{B}$) is potentially true if its argument potentially takes on values whose conjunction is equal to true. However, the operations on $\mathbb{B}$ no longer form a Boolean algebra within either $\mathbf{FSpc}_{nd}$ or $\mathbf{Fspc}_p$:

Proposition 5.2. *The space $\mathbb{B}$ does not form a Boolean algebra (with its usual operations) within either $\mathbf{FSpc}_{nd}$ or $\mathbf{Fspc}_p$.*

Proof. In particular, there is the nondeterministic value both : $* \to_{nd} \mathbb{B}$ and the partial value neither : $* \to_p \mathbb{B}$ satisfying

$$\text{both}^*(\cdot = \text{true}) = \text{both}^*(\cdot = \text{false}) = \top$$
$$\text{neither}^*(\cdot = \text{true}) = \text{neither}^*(\cdot = \text{false}) = \bot,$$

which implies that

$$\text{both} \,||\, !\,\text{both} = \text{both} \neq \text{true}$$
$$\text{neither} \,||\, !\,\text{neither} = \text{neither} \neq \text{true},$$

whereas in a Boolean algebra there is the identity $x \,||\, !\,x = \text{true}$. □

Though $\mathbb{B}$ does not form a Boolean algebra in $\mathbf{FSpc}_{nd,p}$, it does form a *quasi-Boolean algebra*:

Theorem 5.3. *The space $\mathbb{B}$ forms a quasi-Boolean algebra (or De Morgan algebra) in $\mathbf{FSpc}_{nd,p}$, meaning that $\mathbb{B}$ with* &&, ||, true, *and* false *forms a bounded distributive lattice, and* ! *is a De Morgan involution, in that it satisfies $!\,!\,x = x$ and $!(x \,\&\&\, y) = !\,x \,||\, !\,y$.*

Proof sketch. It is instructive to observe how the operations act on generalized points $\Gamma \to_{nd,p} \mathbb{B}$; we will use their equivalent representation as pairs of opens of Γ. Observe that

$$(P_1, Q_1) \,\&\&\, (P_2, Q_2) = (P_1 \wedge P_2, Q_1 \vee Q_2)$$
$$(P_1, Q_1) \,||\, (P_2, Q_2) = (P_1 \vee P_2, Q_1 \wedge Q_2)$$
$$!(P, Q) = (Q, P)$$
$$\text{true} = (\top, \bot) \qquad \text{false} = (\bot, \top).$$

We can use these equations to confirm the various laws, for instance, $!\,!(P, Q) = !(Q, P) = (P, Q)$. □

The argument also shows that $\mathbb{B}$ is a quasi-Boolean algebra in $\mathbf{FSpc}_{nd}$ and $\mathbf{FSpc}_p$ as well. We will show that in each variant, it is possible to quantify these approximate decision procedures over compact-overt spaces.

5.1 Quantification over compact-overt spaces

When working with sets, if a predicate P on a set A is decidable and if A is finite, then $\forall a : A.\ P(a)$ and $\exists a : A.\ P(a)$ are decidable. The spatial analogue of the finite sets is the compact-overt spaces.

A space Σ, called the Sierpiński space, is useful in describing the logic of opens: there is a correspondence $\mathcal{O}(A) \cong A \to_c \Sigma$ between opens of A and Σ-valued continuous maps on A for any space A. We can use this to describe opens via Σ-valued continuous maps. We use the notation $\{x : A \mid e\}$ where e is a Σ-valued term that may mention x, i.e., $x : A \vdash e : \Sigma$, to denote the open subspace $\{A \mid [\![e]\!]\}$. For instance, we can define the open subspace $\{x : \mathbb{R} \mid x \times x < 2\}$, where $(<) : \mathbb{R} \times \mathbb{R} \to_c \Sigma$. We will readily conflate opens and Σ-valued continuous maps, implicitly converting between the two.

5.1.1 On compact-overt spaces

Definition 5.4 (Vickers 1997)**.** A space K is *compact* if for every space Γ, the functor $- \times \top_K : \mathcal{O}(\Gamma) \to \mathcal{O}(\Gamma \times K)$ has a right adjoint $\forall_K : \mathcal{O}(\Gamma \times A) \to \mathcal{O}(\Gamma)$[10]. Similarly, a space A is *overt* if for every space Γ, $- \times \top_A$ has a left adjoint $\exists_A : \mathcal{O}(\Gamma \times A) \to \mathcal{O}(\Gamma)$. A space is *compact-overt* if it is compact and overt.

These conditions are the definitions of universal and existential quantification in terms of adjoints, viewing Γ as some context and opens as truth values in a context. These adjunctions allow us to define syntax for quantification of Σ-valued continuous maps on compact-overt spaces:

$$\frac{\Gamma, x : K \vdash e : \Sigma \qquad K \text{ compact}}{\Gamma \vdash \forall x \in K.\ e : \Sigma} \qquad \frac{\Gamma, x : A \vdash e : \Sigma \qquad A \text{ overt}}{\Gamma \vdash \exists x \in A.\ e : \Sigma}$$

For any compact-overt space K, for any Γ and $P, Q : \mathcal{O}(\Gamma \times K)$, we have in Γ [Vickers 1997]

$$\forall_K(P \vee Q) \le \forall_K P \vee \exists_K Q \quad \text{and} \quad \forall_K P \wedge \exists_K Q \le \exists_K(P \wedge Q).$$

These properties allow us to quantify over compact-overt spaces, too. That is, we can add some syntax

$$\frac{\Gamma, x : K \vdash_C e : \mathbb{B} \qquad K \text{ compact-overt}}{\Gamma \vdash_C \forall x \in K.\ e : \mathbb{B}}$$

$$\frac{\Gamma, x : K \vdash_C e : \mathbb{B} \qquad K \text{ compact-overt}}{\Gamma \vdash_C \exists x \in K.\ e : \mathbb{B}}$$

that behaves as we would expect (for a quasi-Boolean algebra, at least). We interpret this syntax by defining quantification functionals of the type $(\Gamma \times K \to_C \mathbb{B}) \to (\Gamma \to_C \mathbb{B})$. For a compact-overt space K, we define a universal-quantification functional

$$\forall_K : (\Gamma \times K \to_C \mathbb{B}) \to (\Gamma \to_C \mathbb{B})$$
$$\forall_K(P, Q) \triangleq (\forall_K P, \exists_K Q).$$

We confirm this definition works for $C = nd$ because it preserves covering: if $\top \le P \vee Q$, then

$$\top_\Gamma \le \forall_K(\top_{\Gamma\times K}) \le \forall_K(P \vee Q) \le \forall_K P \vee \exists_K Q.$$

Dually, it works for $C = p$ since it preserves disjointness: if $P \wedge Q \le \bot$, then

$$\forall_K P \wedge \exists_K Q \le \exists_K(P \wedge Q) \le \exists_K(\bot_{\Gamma\times K}) \le \bot_\Gamma.$$

We can similarly define the existential-quantification functional by $\exists_K(P, Q) \triangleq (\exists_K P, \forall_K Q)$.

We make the notion that these operations are quantifiers precise by showing that these quantification functionals are adjoints to a weakening functional. The quasi-Boolean algebra on $\mathbb{B}$ determines the preorder which we call *truth order* on maps $A \to_{nd,p} \mathbb{B}$: representing maps as pairs of opens, we define $(P_1, Q_1) \leqq (P_2, Q_2)$ if and only if both $P_1 \le P_2$ and $Q_2 \le Q_1$.

For any spaces Γ and A, weakening $((- \circ \text{fst}) : (\Gamma \to_{nd,p} \mathbb{B}) \to (\Gamma \times A \to_{nd,p} \mathbb{B}))$ is monotone with respect to truth order. The quantifiers deserved to be called such:

[10] This definition of compactness is equivalent to the more common one, that every open cover has a finite subcover.

Theorem 5.5. *The existential- and universal-quantification functionals are left and right adjoints to weakening, respectively, with respect to truth order, i.e.,* $\exists_K \dashv (- \circ \mathrm{fst}) \dashv \forall_K$.

5.1.2 On compact-overt subspaces

Sometimes, the space that we might want to quantify over could depend on some continuous variables in the context. For instance, we may want to quantify a predicate $f : \mathbb{R} \times \mathbb{R} \to_C \mathbb{B}$ over the triangle in $\mathbb{R} \times \mathbb{R}$ bounded by $(0,0)$, $(1,0)$, and $(0,1)$. We will describe a formalism whereby it will be possible to write this as $\forall x \in [0,1].\ \forall y \in [0, 1-x].\ f(x,y)$. We handle this situation by considering spaces whose points represent compact-overt subspaces of some space.

There is a connection between overt spaces and the partiality-and-nondeterminism monad $\mathcal{P}_\Diamond$: Every point of $\mathcal{P}_\Diamond(A)$ corresponds to an overt subspace of A[11]. Similarly, for each space A there is a powerspace $\mathcal{P}_\Box(A)$ whose points correspond with compact subspaces of A [Vickers 2004, 2009]. We summarize its salient characteristics. There is a "necessity" modality $\Box : \mathcal{O}(A) \to \mathcal{O}\left(\mathcal{P}_\Box(A)\right)$ that distributes over meets and directed joins (analogous to the "possibility" modality $\Diamond : \mathcal{O}(A) \to \mathcal{O}\left(\mathcal{P}_\Diamond(A)\right)$ for the lower powerspace). Continuous maps $\Gamma \to_c \mathcal{P}_\Box(A)$ are in bijective correspondence with inverse image maps $\mathcal{O}(A) \to \mathcal{O}(\Gamma)$ that preserve meets and directed joins. Like $\mathcal{P}_\Diamond$, $\mathcal{P}_\Box$ is a strong monad.

The powerspace analogue of the compact-overt spaces is called the *Vietoris powerspace* $\mathcal{P}_{\Box\Diamond}$ [Vickers 1989]. Points of $\mathcal{P}_{\Box\Diamond}(A)$ correspond to compact-overt subspaces of A. The space $\mathcal{P}_{\Box\Diamond}$ has both the possibility and necessity modalities that interact exactly as with the compact-overt spaces.

We can add some additional syntax to make it easier to describe opens with these modalities:

$$\frac{\Gamma \vdash s : \mathcal{P}_\Diamond(A) \qquad \Gamma, x : A \vdash e : \Sigma}{\Gamma \vdash \exists x \in s.\ e : \Sigma}$$

$$\frac{\Gamma \vdash s : \mathcal{P}_\Box(A) \qquad \Gamma, x : A \vdash e : \Sigma}{\Gamma \vdash \forall x \in s.\ e : \Sigma}\ .$$

This syntax is interpreted using the correspondence $\Sigma \cong \mathcal{P}_\Diamond(*) \cong \mathcal{P}_\Box(*)$ [Townsend 2006] via the strong monadic "bind" operations of $\mathcal{P}_\Diamond$ and $\mathcal{P}_\Box$. Accordingly, we can quantify our Booleans over compact-overt subspaces as well in the same way, implementing the syntax

$$\frac{\Gamma \vdash s : \mathcal{P}_{\Box\Diamond}(A) \qquad \Gamma, x : A \vdash_C e : \mathbb{B}}{\Gamma \vdash_C Qx \in s.\ e : \mathbb{B}}$$

where Q is either $\forall$ or $\exists$. Just as in the case of compact-overt spaces, these definitions preserves both covering and disjointness, so C can have any combination of partiality and nondeterminism.

Compact-overt subspaces form a convenient class of spaces over which exhaustive reasoning is possible. The continuous image of a compact-overt space is compact-overt (just as the image of a finite set under any map is finite). Like finite subsets, compact-overt subspaces are closed under finitary union but not necessarily intersection. Naturally, a finite set viewed as a discrete space is compact-overt.

[11] Specifically, the points of $\mathcal{P}_\Diamond(A)$ are in bijective correspondence with the *weakly closed* overt subspaces of A [Vickers 2007b, Theorem 32].

6 Implementation in Marshall

We implemented a pattern-matching construct as well as a library for partial and/or nondeterministic decision procedures within the Marshall programming language for exact real arithmetic [Bauer 2008], which is based on Abstract Stone Duality, a related, though different, theory of constructive topology.

Marshall's type system includes **`real`** ($\mathbb{R}$), **`prop`** (Σ), finitary products, and function types. Notably, it lacks discrete types such as $\mathbb{B}$ and has no support for subspace types. However, we used **`prop * prop`** to simulate $\mathbb{B}$, using the correspondence with pairs of opens described in Proposition 5.1.

Partiality is intrinsic to Marshall, whether in evaluation of terms of type **`prop`**, or evaluation of real numbers defined by Dedekind cuts where there is a gap between the left and right cuts. In the course of adding a pattern-match construct and computational support for it, we added support for nondeterminism in this manner, which was not previously available. Accordingly, Marshall effectively allows programming in $\mathbf{FSpc}_{nd,p}$.

Our pattern-match construct in Marshall has syntax and is typed as follows:

$$\frac{\forall i \in \{1,\ldots,n\}, p_i : \mathsf{prop} \qquad \forall i \in \{1,\ldots,n\}, e_i : t}{(p_1 \texttt{~>} e_1 \,||\, \ldots \,||\, p_n \texttt{~>} e_n) : t}$$

This is different, and substantially less general, than pattern matching described in §4: there is no variable binding, and only finitely many cases are permitted. Regardless, this construct suffices to enable implementation of the approximate decision procedures of §5, including quantification over those compact-overt spaces that are available in Marshall, which are closed intervals with rational endpoints (and implicitly, their finitary products), as well as to implement simplified versions of examples in §7. The modified version of Marshall and examples are available at https://github.com/psg-mit/marshall-lics/. We describe the semantics of Marshall and its extension in further detail in [Sherman et al. 2018].

7 Case studies

We describe two example tasks that require making decisions based on continuous values, demonstrate how the techniques in this work can be applied to solve them, and implement their solutions in Marshall. In §7.1, a car must decide whether to cross an intersection. This example shows the necessity of nondeterminism in decision-making as well as how to reason about these computations. Example §7.2 uses the formal logic from §5 for approximate root-finding.

7.1 Autonomous car approaching a yellow light

Sometimes, partiality is unacceptable. Consider an autonomous car that is approaching a traffic light that has just turned yellow. To ensure safety, the car must be outside of the intersection when the light turns red. This requires the discrete decision to be made of whether or not to proceed through the intersection. We will model this problem with additional concrete detail and demonstrate that it is impossible to do so deterministically, but with nondeterminism it is possible to write a program with a formal safety guarantee.

We model the car's state when the light turns yellow as a position and velocity, $\mathrm{CarState} \triangleq \{(x,v) : \mathbb{R} \times \mathbb{R} \mid 0 < v < v_{\max}\}$, where the velocity is positive but bounded. At the moment the light turns yellow, the car may choose a constant acceleration in the range $\mathrm{Accel} \triangleq \{\mathbb{R} \mid a_{\min} < \cdot < a_{\max}\}$, limited by the car's physical capabilities (with $a_{\min} < 0 < a_{\max}$). The continuous map pos :

$\mathsf{CarState} \times \mathsf{Accel} \to_c \mathbb{R}$ computes where the car will lie when the light turns red, given the car's state when the light turns yellow and the chosen acceleration in the intervening period[12]. We define the position 0 to be where the intersection begins and let $w > 0$ mark the end of the intersection. Thus, we define $\mathsf{safe} : \mathcal{O}(\mathbb{R})$ by $\mathsf{safe} \triangleq (\cdot < 0) \vee (w < \cdot)$.

The problem of choosing an acceleration to safely navigate the intersection is that of finding a function $f : \mathsf{CarState} \to_c \{y : \mathsf{CarState} \times \mathsf{Accel} \mid \mathsf{safe}(\mathsf{pos}(y))\}$ such that $\mathsf{fst} \circ f = \mathsf{id}$.

Proposition 7.1. *It is impossible to continuously choose an acceleration to safely navigate the intersection (i.e., there is no continuous map f as described above).*

Proof. Note that $\{y : \mathsf{CarState} \times \mathsf{Accel} \mid \mathsf{safe}(\mathsf{pos}(y))\}$ has two connected components, corresponding to whether the car is before the intersection ($\mathsf{pos}(y) < 0$) or past the intersection ($w < \mathsf{pos}(y)$) when the light turns red. If there were such an f, then since CarState is connected, so would be the image of f, meaning that f must, regardless of the initial car state, always make the same decision of whether to stop for the intersection or proceed through. But if the car's initial state is sufficiently far back from the intersection, it could not choose an acceleration that ensures it is past the intersection when the light turns red. Conversely, if the car is already past the intersection, it cannot go backwards and thus cannot ensure it is before the intersection when the light turns red. □

Proposition 7.2. *However, if we permit nondeterminism, we* can *produce a map $f : \mathsf{CarState} \to_{nd} \{y : \mathsf{CarState} \times \mathsf{Accel} \mid \mathsf{safe}(\mathsf{pos}(y))\}$ that nondeterministically chooses an acceleration that is always safe, assuming some conditions on the constants a_{min}, a_{max}, v_{max}, w, and the time T between when the light turns yellow and when it turns red.*

Proof. We outline one possible solution. Let $\varepsilon > 0$ be some buffer distance. We can compute as a function of the initial car state the necessary acceleration to proceed through the light and be at position $w + \varepsilon$ when the light turns red, as well as the necessary deceleration to stop before the light at position $-\varepsilon$,

$$\begin{aligned} a_{\mathrm{go}} &: \mathsf{CarState} \to_c \mathbb{R} \\ a_{\mathrm{go}}(x, v) &\triangleq \max\left(0, 2(w + \varepsilon - x - vT)/T^2\right) \\ a_{\mathrm{stop}} &: \mathsf{CarState} \to_p \mathbb{R} \\ a_{\mathrm{stop}}(x, v) &\triangleq \frac{v^2}{2(x + \varepsilon)}. \end{aligned}$$

Note that $a_{\mathrm{go}}(x, v)$ is always nonnegative, and $a_{\mathrm{stop}}(x, v)$ is always nonpositive. We assemble the final solution as

$$f : \mathsf{CarState} \to_{nd} \{y : \mathsf{CarState} \times \mathsf{Accel} \mid \mathsf{safe}(\mathsf{pos}(y))\}$$

$$f(s) \triangleq \mathsf{case}(s) \begin{cases} \iota[\lambda c.\ a_{\mathrm{go}}(c) < a_{\max}](c') & \Rightarrow (c', a_{\mathrm{go}}(c')) \\ \iota[\lambda c.\ a_{\min} < a_{\mathrm{stop}}(c)](c') & \Rightarrow (c', a_{\mathrm{stop}}(c')) \end{cases}.$$

Formal proof would be required to show that the output of each branch is indeed within the required subspaces indicated by the output type.

[12] However, we enforce that velocity remains nonnegative, so if the car decelerates to zero velocity before the light turns red, it remains stopped rather than going backwards.

```
let a_go = fun x : real => fun v : real =>
    max 0 (2 * (w + eps - x - v * T) / (T * T));;
let a_stop = fun x : real => fun v : real =>
    v * v / (2 * (x + eps));;
let accel = fun x : real => fun v : real =>
  (  a_go x v   < a_max  ~>  a_go x v
  || a_stop x v > a_min  ~>  a_stop x v );;
```

Figure 4. A Marshall program that uses an overlapping pattern match to nondeterministically compute the desired acceleration of an autonomous car approaching a traffic light.

With sufficient conditions on the constants, we can prove that the cases of f are covering, i.e., that it is always the case that either the go or stop strategy is applicable[13]. □

This translates to the Marshall program in Fig. 4.

7.2 Approximate root-finding

Given any continuous function $f : K \to_c \mathbb{R}$, where K is compact-overt, then least one of the following statements must hold:

- There is some $x \in K$ such that $|f(x)| < \varepsilon$.
- For every $x \in K$, $f(x) \neq 0$.

The following root-finding program nondeterministically computes which statement holds, in the former case computing some $x \in K$ that is almost a root:

$$\mathsf{roots}_f : * \to_{nd} \{* \mid \forall x \in K.\ f(x) \neq 0\} + \{x : K \mid |f(x)| < \varepsilon\}$$

$$\mathsf{roots}_f \triangleq \mathsf{case}(\mathsf{tt}) \begin{cases} \iota[\exists x \in K.\ |f(x)| < \varepsilon](y) & \Rightarrow \mathsf{inr}(\mathsf{simulate}(y)) \\ \iota[\forall x \in K.\ f(x) \neq 0](n) & \Rightarrow \mathsf{inl}(n) \end{cases}.$$

That the two cases cover follows from the logic in §5: the opens $|\cdot| < \varepsilon$ and $\cdot \neq 0$ cover $\mathbb{R}$, and covering opens are stable under pullback by continuous maps f and quantification over compact-overt spaces K.

It remains to define simulate, which in general has the type

$$\mathsf{simulate} : \{* \mid \exists x \in A.\ U(x)\} \to_{nd} \{x : A \mid U(x)\}$$

for any overt space A and open U of A (in roots_f, A is K and U is $\lambda x.\ |f(x)| < \varepsilon$). Given the existence of some values that satisfy a property U of A, simulate can nondeterministically simulate those values. It is defined by the inverse image map

$$\begin{aligned} \mathsf{simulate}^* &: \mathcal{O}(\{A \mid U\}) \to \mathcal{O}(\{* \mid \exists_A U\}) \\ \mathsf{simulate}^*(V) &\triangleq \exists_A (V \wedge U). \end{aligned}$$

The program roots_f accomplishes the task of approximate root-finding over a very general class of functions with a very short definition that works by composing constructs from §4 and §5. That K is compact-overt means that it implements a general computational interface for exhaustive search.

The Marshall functional in Fig. 5 approximately decides whether a real-valued function $f : \mathbb{R} \to_c \mathbb{R}$ has roots on the interval $[0, 1]$.

[13] The cases cover so long as $w + 2\varepsilon \leq \frac{1}{2} T^2 a_{\max}$, i.e., it is possible to speed up from a standstill to cross the intersection, and

$$v_{\max} < -a_{\min}\left(T + \sqrt{T^2 - 2\left(\tfrac{1}{2} T^2 a_{\max} - w - 2\varepsilon\right)/a_{\min}}\right)$$

which guarantees that the car never goes so fast that it is too close to stop and also cannot speed up to pass through the intersection in time.

```
let exists_bool_interval = fun pred : real -> bool =>
     (exists x : [0,1], is_true  (pred x)) ~> tt
  || (forall x : [0,1], is_false (pred x)) ~> ff ;;
let is_0_eps = fun x : real =>
       x < 0 \/ x > 0   ~> ff
  || -eps < x /\ x < eps ~> tt ;;
let roots_interval = fun f : real -> real =>
  exists_bool_interval (fun x : real => is_0_eps (f x));;
```

Figure 5. A Marshall program that approximately computes whether a continuous function f has roots on $[0, 1]$.

8 Related work

Several works exploit the ability to quantify over compact-overt spaces [Escardó 2004, 2007; Simpson 1998; Taylor 2010] to compute values in $\mathbb{R}$, $\mathbb{B}$, or Σ, but not partial or nondeterministic Booleans.

$\mathbf{FSpc}_{nd,p}$ is equivalent to the category of suplattices (also known as complete join semilattices), which is well-studied and its relation to **FSpc** well-documented [Ciraulo et al. 2013; Johnstone 2002; Townsend 2006; Vickers 2004]. Vickers [1989] defines $\cdot_\perp$ and $\mathcal{P}^+_\Diamond$, but we are not aware of any previous explicit characterizations of these constructions as strong monads induced by adjoints to the forgetful functors to $\mathbf{FSpc}_p$ and $\mathbf{FSpc}_{nd}$, respectively.

Several works describe related programming formalisms involving continuity, partiality, and nondeterminism. Marcial-Romero and Escardó [2007] define a language with real-number computation that admits nontermination and has a foundational family of functions $\mathrm{rtest}_{a,b}$ which map real numbers to nondeterministic Boolean values, with a domain-theoretic semantics that uses Hoare powerdomains (which roughly corresponds to $\mathcal{P}_\Diamond$). Establishing totality requires reasoning within their operational model, in contrast to our framework, which optionally has denotational semantics for total functions. Escardó [1996] defines a language "Real PCF" with a denotational semantics in terms of cpo's, in which there is an operation known as a "parallel conditional," which corresponds to the internal "or" operation on the Sierpiński space $\vee_\Sigma : \Sigma \times \Sigma \to_c \Sigma$ in our formalism. Parallel conditionals are applied to construct deterministic functions on the real numbers, which differs from our examples, whose computations are total but nondeterministic. Similarly, Tsuiki's work on computation with Gray-code-based real numbers 2002 is based on "indeterministic" computation, where potentially nonterminating computations must be interleaved, and those that terminate must agree in their answers.

We are unaware of any other notion of pattern matching that permits patterns where determining membership is undecidable, without jeopardizing totality. Müller [2009] describes a system for exact real arithmetic that has a datatype of "lazy Booleans" analogous to our partial Booleans, as well as a partial and nondeterministic n-ary choose operation on lazy Booleans. *dReal* is a tool that allows computation of approximate truth values over $\mathbb{R}$ [Gao et al. 2012], allowing order comparisons and bounded quantifiers. Our calculus of nondeterministic $\mathbb{B}$-valued maps, when restricted to $\mathbb{R}$, provides similar computational abilities but with a different foundational framework.

9 Conclusion

We presented a semantic framework for principled computation with continuous values with partiality and/or nondeterminism. In each variant, pattern-matching constructs facilitate construction of programs, and the Booleans yield a formal logic for approximate decision procedures. The programs we describe are executable, thanks to their use of constructive topology, as demonstrated by their implementation in our modified version of Marshall.

Acknowledgments

We thank Tej Chajed, Gabriel Scherer, and Muralidaran Vijayaraghavan for their feedback on drafts. This work was supported in part by the US Department of Energy (Grants DE-SC0014204, DE-SC0008923) and the Office of Naval Research (Grant N00014-17-1-2699).

References

Abhishek Anand and Ross A Knepper. 2015. ROSCoq: Robots Powered by Constructive Reals. *Interactive Theorem Proving* 15, 15 (2015), 2015.

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

Errett Bishop. 1967. *Foundations of Constructive Analysis.* Academic Press.

Francesco Ciraulo, Maria Emilia Maietti, and Giovanni Sambin. 2013. Convergence in formal topology: a unifying notion. *Journal of Logic and Analysis* 5 (2013).

Thierry Coquand, Giovanni Sambin, Jan Smith, and Silvio Valentini. 2003. Inductively generated formal topologies. *Annals of Pure and Applied Logic* 124, 1 (2003), 71–106.

Martín Escardó. 1996. PCF extended with real numbers. *Theoretical Computer Science* 162, 1 (1996), 79–115.

Martín Escardó. 2004. Synthetic topology: of data types and classical spaces. *Electronic Notes in Theoretical Computer Science* 87 (2004), 21–156.

Martín Escardó. 2007. Infinite sets that admit fast exhaustive search. In *22nd Annual IEEE Symposium on Logic in Computer Science.* IEEE, 443–452.

Sicun Gao, Jeremy Avigad, and Edmund M Clarke. 2012. Delta-decidability over the reals. In *27th Annual IEEE Symposium on Logic in Computer Science.* IEEE, 305–314.

Peter T Johnstone. 1977. *Topos theory.* Academic Press.

Peter T Johnstone. 2002. *Sketches of an elephant: A topos theory compendium.* Vol. 2. Oxford University Press.

Saunders Mac Lane and Ieke Moerdijk. 1992. *Sheaves in Geometry and Logic: A First Introduction to Topos Theory.* Springer-Verlag.

J. Raymundo Marcial-Romero and Martín Escardó. 2007. Semantics of a sequential language for exact real-number computation. *Theoretical Computer Science* 379, 1-2 (2007), 120–141.

Norbert Th Müller. 2009. Enhancing imperative exact real arithmetic with functions and logic. In *Computability and Complexity in Analysis, Ljubljana.*

Russell O'Connor. 2008. Certified exact transcendental real number computation in Coq. In *Theorem Proving in Higher Order Logics.* Springer, 246–261.

Erik Palmgren. 2003. Predicativity problems in point-free topology. In *Proceedings of the Annual European Summer Meeting of the Association for Symbolic Logic, held in Helsinki, Finland.* 221–231.

Benjamin Sherman, Luke Sciarappa, Adam Chlipala, and Michael Carbin. 2018. Computable decision making on the reals and other spaces via partiality and nondeterminism. arXiv:1805.00468

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

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

Christopher F Townsend. 2006. On the parallel between the suplattice and preframe approaches to locale theory. *Annals of Pure and Applied Logic* 137, 1-3 (2006), 391–412.

Hideki Tsuiki. 2002. Real Number Computation Through Gray Code Embedding. *Theor. Comput. Sci.* 284, 2 (July 2002), 467–485.

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

Steven Vickers. 1997. Constructive points of powerlocales. In *Mathematical Proceedings of the Cambridge Philosophical Society*, Vol. 122. Cambridge University Press, 207–222.

Steven Vickers. 2004. The double powerlocale and exponentiation: a case study in geometric logic. *Theory and Applications of Categories* 12, 13 (2004), 372–422.

Steven Vickers. 2005. Localic completion of generalized metric spaces I. *Theory and Applications of Categories* 14, 15 (2005), 328–356.

Steven Vickers. 2007a. Locales and toposes as spaces. In *Handbook of spatial logics.* Springer, 429–496.

Steven Vickers. 2007b. Sublocales in formal topology. *The Journal of Symbolic Logic* 72, 02 (2007), 463–482.

Steven Vickers. 2009. The connected Vietoris powerlocale. *Topology and its Applications* 156, 11 (2009), 1886–1910.

K. Weihrauch. 1995. *A simple introduction to computable analysis.* Fernuniv., Fachbereich Informatik.