Formalization of Equational Reasoning in the Set-Theoretic Interpretation of Type Theory

Takafumi Saikawa

April 6, 2020

Contents

1 Introduction **9**

2 Background: Formalization through the Set-Theoretic Interpretation of Type Theory **13**

2.1 Introduction . . . 13
2.2 Basic layer of elementary mathematics – first-order logic . . . 13
2.2.1 Syntax . . . 14
2.2.2 Semantics . . . 18
2.3 Zermelo-Fraenkel set theory . . . 20
2.4 Type Theory . . . 24
2.4.1 Calculus of constructions . . . 25
2.4.2 Calculus of inductive constructions . . . 27
2.5 Set-theoretic semantics . . . 27
2.5.1 Grothendieck universe . . . 28
2.6 Formalization techniques in Coq . . . 28
2.6.1 Type inference and canonical structures . . . 29
2.6.2 Packed classes . . . 29

3 Formalization of Convex Spaces **33**

3.1 Introduction . . . 33
3.2 Convex spaces . . . 34
3.3 Multiary convex combination . . . 35
3.4 Conical spaces and embedded convex spaces . . . 37
3.5 Formalization of convex functions . . . 40
3.6 Applications in information theory . . . 41
3.7 Nonempty convex sets – application to program semantics . . . 44
3.8 Related work . . . 46

4 Formalization of Computational Effects (Equational Reasoning on Monads) 47
4.1 Introduction 47
4.2 Notions from category theory 48
4.3 Formalization of the theories of functors and monads 50
4.4 Extensions of the theory of monads 53
4.4.1 Monads with nondeterministic choice 53
4.4.2 Monads with probabilistic choice 54
4.4.3 Monads with combined choice 55
4.5 Examples of equational reasoning 56
4.5.1 rev_insert (taken from [Mu, 2019]) 56
4.5.2 two_coins 58
4.5.3 arbcoin and coinarb (taken from [Gibbons and Hinze, 2011]) 59
4.6 Models 61
4.6.1 Semilattice model for `MonadAltCI` [Affeldt et al., 2020a, `monad_model.v`] 61
4.6.2 Distribution model for `MonadProb` [Affeldt et al., 2020a, `proba_monad_model.v`] 61
4.6.3 Nonempty convex powerset model for `MonadAltProb` 62

5 Equational Reasoning on Categories 63
5.1 Introduction 63
5.2 Formalization of Categories 64
5.3 Examples of formalized categories 66
5.3.1 Category of sets 66
5.3.2 Category of sets with choice functions 66
5.4 Affine functions and the category of convex spaces 66
5.5 Formalization of monads defined from adjunctions 67
5.6 Semicomplete semilattice 67
5.7 Semicomplete semilattice convex space 68
5.7.1 Definition 68
5.7.2 Technical lemmas 69
5.7.3 Category of semicomplete semilattice convex space 69
5.8 Construction of a model for the combined choice 70
5.9 Related work 71
5.9.1 Some history 71
5.9.2 Comparison to Cheung's finitely generated model 71
5.9.3 Comparison to domain-theoretic combination of nondeterminisms 71
5.9.4 Comparison to other formalizations of category theory 72

5.9.5 Category of convex spaces 72

Acknowledgements

First of all, I would like to sincerely thank my advisor Jacques Garrigue for his kind and careful attention to every part of my work. Through many discussions, his bird's-eye view pinpointed where the interesting ideas are buried, and directed me to the way to properly present them. Without his continuous support, this thesis would not have been finished.

I would like to thank Reynald Affeldt for having always encouraged me and cranked up my research. His invitation to me to be his academic colleague ignited a strong motivation towards this work. Without his words, this thesis would not have been initiated.

My work on formalization started when I collaborated with the JSPS KAKENHI project "Formalization on Modern Coding Theory". I learned there that formalization would be a serious mathematical activity in the near future. For that cheerful opportunity, I would like to thank the investigators of the project Manabu Hagiwara, Kenta Kasai, and Shigeaki Kuzuoka.

I have been greatly supported by many friends during the writing of this thesis. I especially acknowledge the contribution by Kazunari Tanaka, in which he solved a complicated dependent-type puzzle in the formalization of categories.

Finally, I would like to thank my family. Everything in this thesis is built upon their love.

Chapter 1

Introduction

The method of equational reasoning is a style for mathematical proofs which is characterized by heavy uses of rewriting along equations. Not only values but propositions are rewritten using equivalence lemmas.

This style is common in proofs of pure mathematics but not necessarily so in proofs written using computerized proof assistants. Proofs assistants are software that aid human users to write and check mathematical proofs. They are based on systems of formal logic, in which proofs are expressed as tree structures. Using such representations of proofs, it is easier for the users of proof assistants to proceed with a deductive style of proof. This tendency is strengthened by the algorithmic support for writing such proofs (semi-)automatically.

Mathematical proofs have also been used to prove properties of computer programs, assuring their correctness. Since programs naturally have inductively defined structures in both the explicit syntax and implicit semantics, proofs on programs tend to rely on induction everywhere.

While induction can be thought of as a variant of deductive proofs, equational reasoning in program proofs became a popular idea with the emergence of functional programming languages in late 1980s [Bird, 1987, Bird, 1989].

Equational reasoning fits very well to *pure* functional programs, whose every part has the *referential transparency*, meaning that such parts of pure programs can be replaced with some other pieces of programs with the same meaning (or extension), without changing the meaning of the whole program. This is the property that enabled equational reasoning on functional programs.

However real-world programs have to interact with the outer environment through *effects*. If they do not have any effect on the environment, one cannot even observe them. The properties of effects, however, are far from pure; it is the essence of effects to break purity. Yet it was a common belief that equational reasoning could not be applied to effectful programs.

In the same age, there was a revelation of the use of monads in the programming language se-

mantics [Moggi, 1989] and functional programming [Wadler, 1990]. Monads enabled the separation of effects from pure parts of programs. Moreover the separated effects can be given an algebraic structure which serves as interface to the effects [Plotkin and Power, 2001].

Equational reasoning on effects through such algebraic interface was first seriously considered by Gibbons and Hinze [Gibbons and Hinze, 2011]. They employed Plotkin and Power's idea in their reasoning on effects: the theory of monads extended by algebraic operations of an effect captures the necessary properties of that effect.

Our work is based upon these foundations. We formalize using a proof assistant various aspects of equational reasoning. Especially we focus on the formalization of monadic equational reasoning on effects. The formalization includes both theories which are applied to proofs on concrete programs, and models which assure the consistency of those theories.

We also extend the method of equational reasoning by developing a framework for category theory. This framework features the use of concrete categories for lightweight usability. It enables the shallow embedding of programs whose properties can only be captured at categorical level. As a specific example, we construct the first formalized model of the theory of combined probabilistic and nondeterministic choice.

This construction of model is backed by the formalization of convex spaces. Convex spaces appear in various domains of pure and applied mathematics. Although it is also an equational theory, it is poor in algebraic properties and does not allow easy rewriting. We investigated its connection to conical spaces, which have proper linearity, and devised a method to prove properties of convex spaces in conical spaces. This turned out to be useful in many parts of our work.

My contributions can be summarized as several pieces of work on formalization. The examples include various instances of convex spaces, ordered convex spaces and a generalized form of convex functions, semicomplete semilattices with a Coq-friendly axiomatization, nonempty convex powerset and its semicomplete semilattice convex space structure, a shallow-embedding of category theory using concrete categories, and a model of the combined choice.

Each formalization involves a process of carefully choosing the correct forms of axioms among logically equivalent formulations. These design choices relies on a careful inspection into the detail of each theory, leading often to conceptual findings about the theory. It is often required for a successful formalization to introduce new forms of axiomatization and reorganize existing theories. In this thesis, it is exemplified by our axiomatization of semicomplete semilattices (Section 5.6). Another example is the identification that shallow-embedding categories turned out to be concrete categories. As far as I know, this use of concrete categories as a means of shallow-embedding categories is first utilized in our formalization.

Various new technical lemmas are proved according to these design choices. These lemmas

subsume those present in the literature (see Section 5.7.2).

The outline of the remaining chapters is as follows.

Chapter 2 reviews the mathematical background that is referred to throughout this thesis.

Chapter 3 explains the notion of convex space and its formalization. The method of packed classes is used to define the theory of convex spaces and a hierarchy extending it.

Chapter 4 deals with the formalization of effects in computer programs. The effects are presented as the category-theoretic notion of monads. We explain how the packed classes are extended to be able to handle monads over the category of sets.

Chapter 5 goes further along the line of the previous chapter by formalizing the notion of category other than that of set. The motivating example is the problem of combining probabilistic and nondeterministic choices. We apply our formalization of category theory to construct a model of the combined choice.

Chapter 2

Background: Formalization through the Set-Theoretic Interpretation of Type Theory

2.1 Introduction

Our definitions and theorems are presented in one or both of two versions: informal and formalized. Informal versions are going to be founded on the classical Zermelo-Fraenkel set theory (ZF). On the other hand, formalization proceeds in the Calculus of Inductive Constructions (CIC), a powerful variant of type theory implemented as the proof-assistant Coq. The connection between these two theories is given by the set-theoretic semantics, which interprets CIC in ZF. The purpose of this chapter is to review these theories and semantics, fixing notations. An experienced reader can skip this whole chapter, only needing to occasionally come back here to check the notations and foundational settings.

2.2 Basic layer of elementary mathematics – first-order logic

In this section, we review the syntax and semantics of first-order logic (FOL), which we need to define the Zermelo-Fraenkel set theory in the next section.

First-order logic also provides good examples of metamathematical notions and formalism, which are shared with the formalization in type theory. We dig into a bit of details of first-order logic in the rest of this section, in order to prepare the metamathematical language and show the similarities of the informal mathematics to the formalized ones.

2.2.1 Syntax

Formal theories are defined in two parts: syntax and semantics. The syntax of a theory describes mathematical objects usually in finite words. The semantics relates the descriptions to actual mathematical objects. We start with briefly reviewing the syntax of first-order logic.

Language

A (first-order) *language* is a collection of symbols. Each symbol is either a *function symbol* or a *predicate symbol.* We associate to each symbol a natural number, called the *arity* of the symbol. A function (or predicate) symbol with arity n is called an n*-ary* function (resp. predicate) symbol. A nullary[1] function symbol is often called a *constant* symbol.

We often denote a language by $\mathcal{L}$.

Examples of languages

Let us enumerate a few examples of languages.

Rings Ring theory is an abstraction of the properties of integers and other similar structures with addition and multiplication. The language of ring theory is given as follows.

function symbols $\{\mathtt{0}, \mathtt{1}, -, +, \cdot\}$.

predicate symbols None.

arity $\mathtt{0}$ and $\mathtt{1}$ are nullary, $-$ unary, and $+$ and $\cdot$ binary.

Ordered rings Often a ring has an ordering over it. The ring of integers is one such[2] . We add a symbol to the language of ring theory in order to express the statements about ordering. The language of the theory of ordered rings is thus as follows.

function symbols Same as the language of ring theory.

predicate symbols $\{<\}$.

arity $<$ is binary.

[1] Adjectives 0-ary, 1-ary, and 2-ary are abbreviated as nullary, unary, and binary, respectively.

[2] Think of a statement like $1 < 2$.

Sets Zermelo-Fraenkel set theory (ZF) is the standard axiomatization of the notion of sets. The language of ZF has only one symbol.

function symbols None.

predicate symbols $\{\in\}$.

arity $\in$ is binary.

Logical symbols

In addition to the symbols in a language, we use *logical* symbols to write logical formulas. There are seven logical symbols in our system.

$$\Rightarrow, \neg, \vee, \wedge, \exists, \forall, = .$$

Logical symbols can be used with any language to write sentences in first-order logic. On the other hand, symbols in a language are not necessarily shared with another language, and therefore called *nonlogical* symbols. We assume, without loss of generality, that the set of nonlogical symbols is disjoint to that of logical symbols.

Terms and formulas

Hereafter, we assume that the reader can read Backus-Naur forms (BNFs), and understands that they define sets of trees (*abstract syntax trees*).

Let $\mathcal{L}$ be a language and V be a countable set. We assume that V is disjoint to the set of logical or nonlogical symbols. We call the elements of V *variables*.

A *term* of $\mathcal{L}$ is a tree constructed from function symbols and variables. The definition is given by the following BNF. Note that we treat parentheses as structural parts of trees, not symbols.

$$\begin{aligned} \textbf{term} \ni t_1, t_2, \ldots ::= {} & x \qquad \text{(for } x \text{ a variable)} \\ & \mid \mathtt{f}(t_1, \ldots, t_n) \qquad \text{(for } \mathtt{f} \text{ an } n\text{-ary function symbol).} \end{aligned}$$

A *formula* of $\mathcal{L}$ is a tree constructed from logical symbols, predicate symbols, variables, and terms.

The definition is as follows.

$$
\begin{aligned}
\textbf{formula} \ni \phi, \psi ::= {} & \mathtt{p}(t_1, \ldots, t_n) \qquad \text{(for } \mathtt{p} \text{ an n-ary predicate symbol)} \\
& \mid t_1 = t_2 \\
& \mid (\phi) \Rightarrow (\psi) \\
& \mid \neg(\phi) \\
& \mid (\phi) \vee (\psi) \\
& \mid (\phi) \wedge (\psi) \\
& \mid \exists x, (\phi) \qquad \text{(for } x \text{ a variable)} \\
& \mid \forall x, (\phi) \qquad \text{(for } x \text{ a variable).}
\end{aligned}
$$

In writing terms and formulas, we will often drop parentheses as long as it does not introduce ambiguity.

Occurrence

Given a tree, we distinguish a subtree and its *occurrences* in the tree. A subtree is specified by its contents, that is, what it contains as its branches in what order. This is no different from how a tree is specified. The only difference between the notions of tree and subtree is whether we assume the existence of a whole tree containing it. On the contrary, an occurrence of a subtree is specified by where it is in the whole tree. We therefore define an occurrence to be a list of natural numbers, which indicates the path to a subtree from the root. If a number n appears as the k-th element of the list, it corresponds to the choice of n-th branch at a node of depth k.

For example, in the formula

$$(x = 3 \wedge x = 4) \wedge y = 5,$$

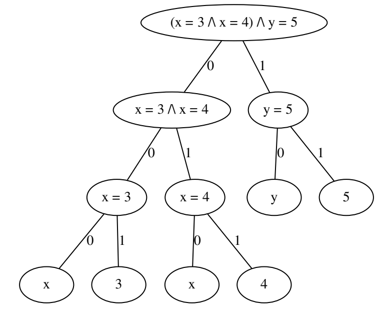
the only occurrence of "$x = 3$" is $\langle 0, 0 \rangle$. There are two occurrences of "x": $\langle 0, 0, 0 \rangle$ and $\langle 0, 1, 0 \rangle$.

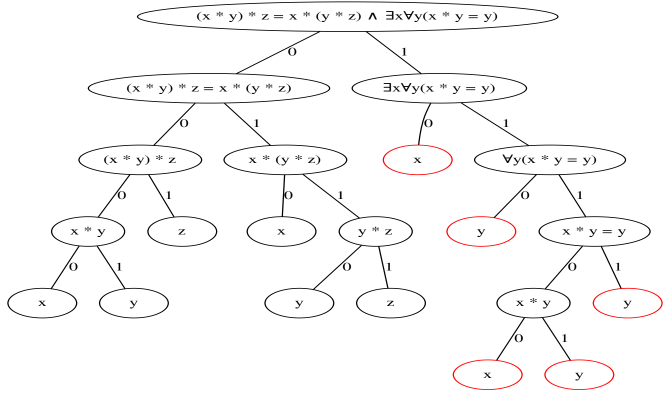
Binding

We say that an occurrence of a variable x is *bound* if it is in an occurrence of a subtree of the form $\exists x, \phi$.[3] Otherwise, the occurrence is *free*. A variable which has a free (or bound) occurrence is called a *free (or respectively, bound) variable*.

Let α be a term or a formula. We denote by $\mathrm{FV}(\alpha)$ the set of all free variables in α. If $\mathrm{FV}(\alpha)$ is empty, α is said to be *closed*.

[3] That is, the path to the occurrence of $\exists x, \phi$ is an initial segment of the specified occurrence of x.





An example: $(x * y) * z = x * (y * z) \wedge \exists x, \forall y, (x * y = y)$.
Bound occurrences of variables are indicated in red.

Substitution

Let ϕ be a formula and t a term. We obtain a new formula $\phi[x := t]$ by substituting t for all free occurrences of x in ϕ. In order to avoid "capturing" free variables, we introduce the following convention: when we write $\phi[x := t]$, we always assume that t is *substitutible for x in ϕ*; that is, for any free variable y in t, ϕ does not contain a subformula of the form $\exists y \psi$ with x free in ψ. Violating this convention leads to a wrong substitution. For example, consider the formula $(\exists y, x < y)[x := y]$ as a statement about real numbers. $\exists y, x < y$ is intuitively true for any choice of x, but the result of the substitution $\exists y, y < y$ is just false. We see later that we can safely rename bound variables to address this sort of difficulties.

Remark. We have taken a detour from the tradition in that terms, formulas and occurrences are defined in terms of trees. See Shoenfield [Shoenfield, 1967] or Kunen [Kunen, 2009] for the traditional approach where terms and formulas are defined as sequences of symbols.

Remark. In the next subsection, we define the semantics to the syntax we have defined so far. The semantics is a classical one, in which some connectives we have introduced in the definition of formulas are not fully needed.

$$\Rightarrow, \vee, \forall.$$

These three connectives are translated to other connectives in the following manner:

$$\begin{aligned}(\phi) \vee (\psi) &:= \neg((\neg(\phi)) \wedge (\neg(\psi))),\\ (\phi) \Rightarrow (\psi) &:= (\neg(\phi)) \vee (\psi),\\ \forall x(\phi) &:= \neg(\exists x(\neg(\phi))).\end{aligned}$$

2.2.2 Semantics

Now we define the standard (Tarskian) semantics of first-order logic. It relates to a first-order language a structure (set with a structure) using set theory. First we give a meaning to each symbol in a language, and then extend it to entire terms and formulas.

Structure

We take the meaning of function/predicate symbols simply to be set-theoretic functions/predicates.

Let $\mathcal{L}$ be a language. A *structure* $\mathfrak{A}$ for $\mathcal{L}$ consists of the following data.

- A nonempty underlying set[4] A.
- For each n-ary function symbol $\mathtt{f}$ of $\mathcal{L}$, a function $\mathtt{f}_{\mathfrak{A}} : A^n \to A$.
- For each n-ary predicate symbol $\mathtt{p}$, a subset $\mathtt{p}_{\mathfrak{A}}$ of A^n.

Assignment

In order to interpret terms and formulas, we have to deal with free variables. Intuitively, free variables mean placeholders that are to be replaced by values in a structure. The following definition makes it precise.

Let $\mathcal{L}$ be a language, $\mathfrak{A}$ be a structure for $\mathcal{L}$ with the underlying set A, and V be the set of variables. An *assignment* is a function from a subset of V to A. If α is a term or a formula of $\mathcal{L}$, an *assignment for* α is an assignment whose domain contains $\mathrm{FV}(\alpha)$.

[4] Also called *carrier set*, *universe* or *domain of discourse* in the literature.

Interpretation of terms and formulas

The meaning of terms and formulas in a language is recursively determined once we fix a structure and an assignment of values to free variables.

Let $\mathcal{L}$ be a language and $\mathfrak{A}$ be a structure for $\mathcal{L}$ with the underlying set A. For a term t and an assignment σ for t, the interpretation $[\![t]\!]_{(\mathfrak{A},\sigma)}$ is an element of A defined as follows.

- If t is a variable x, then
$$[\![t]\!]_{(\mathfrak{A},\sigma)} := \sigma(x).$$
- If t is an $\mathtt{f}(u_1, \ldots, u_n)$ for a function symbol $\mathtt{f}$ and terms $u_1, \ldots, u_n$, then
$$[\![t]\!]_{(\mathfrak{A},\sigma)} := \mathtt{f}_{\mathfrak{A}}([\![u_1]\!]_{(\mathfrak{A},\sigma)}, \ldots, [\![u_n]\!]_{(\mathfrak{A},\sigma)}).$$

For a formula ϕ and an assignment σ for ϕ, the interpretation $[\![\phi]\!]_{(\mathfrak{A},\sigma)}$ is an element of the set $\{\bot, \top\}$ of truth values.

- If ϕ is a $\mathtt{p}(t_1, \ldots, t_n)$ for a predicate symbol $\mathtt{p}$ and terms $t_1, \ldots, t_n$, then
$$[\![\phi]\!]_{(\mathfrak{A},\sigma)} := \begin{cases} \top, \text{ if } \langle [\![t_1]\!]_{(\mathfrak{A},\sigma)}, \ldots, [\![t_n]\!]_{(\mathfrak{A},\sigma)} \rangle \in \mathtt{p}_{\mathfrak{A}} \\ \bot, \text{ otherwise} \end{cases}.$$
- If ϕ is a $t_1 = t_2$, then
$$[\![\phi]\!]_{(\mathfrak{A},\sigma)} := \begin{cases} \top, \text{ if } [\![t_1]\!]_{(\mathfrak{A},\sigma)} = [\![t_2]\!]_{(\mathfrak{A},\sigma)} \\ \bot, \text{ otherwise} \end{cases}.$$
- If ϕ is a $\neg(\psi)$, then
$$[\![\phi]\!]_{(\mathfrak{A},\sigma)} := \begin{cases} \top, \text{ if } [\![\psi]\!]_{(\mathfrak{A},\sigma)} = \bot \\ \bot, \text{ otherwise} \end{cases}.$$
- If ϕ is a $(\psi_1) \wedge (\psi_2)$, then
$$[\![\phi]\!]_{(\mathfrak{A},\sigma)} := \begin{cases} \top, \text{ if } [\![\psi_1]\!]_{(\mathfrak{A},\sigma)} = [\![\psi_2]\!]_{(\mathfrak{A},\sigma)} = \top \\ \bot, \text{ otherwise} \end{cases}.$$
- If ϕ is a $\exists x(\psi)$, then
$$[\![\phi]\!]_{(\mathfrak{A},\sigma)} := \begin{cases} \top, \text{ if there is an element } a \in A \text{ such that } [\![\psi]\!]_{(\mathfrak{A},(\sigma\restriction(\mathrm{dom}(\sigma)\setminus\{x\})\cup\{x\mapsto a\})} = \top \\ \bot, \text{ otherwise} \end{cases}.$$

The last case may need an explanation. We are interpreting the formula $\exists x(\psi)$ as "we can find an example a (out of our underlying set A) for the variable x, such that with a given assignment σ together with the additional assignment of a to x, ψ yields to be true." $(\sigma \upharpoonright (\mathrm{dom}(\sigma) \setminus \{x\}) \cup \{x \mapsto a\}$ is the new assignment with a mapping from x to a added.

Satisfaction and model

Several semantical notions are introduced in this and the next subsection.

Let $\mathcal{L}$ be a language, $\mathfrak{A}$ be a structure for $\mathcal{L}$, and ϕ be a formula of $\mathcal{L}$. If $[\![\phi]\!]_{(\mathfrak{A},\sigma)} = \top$ for any assignment σ for ϕ, we say that

- $\mathfrak{A}$ *models* ϕ, or
- $\mathfrak{A}$ *satisfies* ϕ, or
- ϕ is *valid in* $\mathfrak{A}$,

and denote it by $\mathfrak{A} \vDash \phi$.

Similarly for a set Γ of formulas, we say that $\mathfrak{A}$ is a *model* of Γ if for every formula $\psi \in \Gamma$, $\mathfrak{A} \vDash \psi$ holds; and we denote it by $\mathfrak{A} \vDash \Gamma$. We say that Γ is *satisfiable* if there is a model of Γ.

Semantic consequence

Let $\mathcal{L}$ be a language, $\mathfrak{A}$ be a structure for $\mathcal{L}$. For a formula ϕ and a set Γ of formulas, we say that ϕ is a *semantic consequence* or a *logical consequence* of Γ if for every structure $\mathfrak{A}$ such that $\mathfrak{A} \vDash \Gamma$, $\mathfrak{A} \vDash \phi$ holds. We denote this relation by $\Gamma \vDash \phi$.

Remark. Be careful that we have overloaded the notation $\vDash$ for quite a different uses. $\mathfrak{A} \vDash \phi$ states the relation between a structure and a formula. On the other hand, $\Gamma \vDash \phi$ states the relation between a set of formulas and a formula.

2.3 Zermelo-Fraenkel set theory

So far we have prepared the syntax and semantics of first-order logic. We utilize it here to define Zermelo-Fraenkel set theory (abbreviated ZF set theory) which is the another foundational piece of the informal part of our work.

ZF is a first-order axiomatization of set theory. It is given as a collection of countably many axioms over the language $\{\in\}$. We are going to list axioms and give an explanation, especially to the points which are going to differ from the type theory introduced later. As the intuitive meaning of a term in ZF set theory is always a set, we often use the word "set" to refer to a term.

We often use the following abbreviations of restricted quantification:

$$(\forall x \in y, \phi) \iff (\forall x, x \in y \Rightarrow \phi),$$

$$(\exists x \in y, \phi) \iff (\exists x, x \in y \wedge \phi).$$

Axiom of extensionality

The axiom of extensionality is the following formula:

$$\forall x,\ \forall y,\ (\forall z,\ (z \in x \iff z \in y)) \Rightarrow x = y.$$

This axiom says that sets are compared only by its contents, not by considering how it is created or what it is named.

Axiom schema of separation

For any formula $\phi(\overline{z}, A, x)$, the formula

$$\forall \overline{z},\ \exists X,\ \forall x,\ (x \in X \iff x \in A \wedge \phi)$$

is an axiom. Note that the schema is parametrized by formulas ϕ, and represents countably many axioms.

This axiom schema allows one to form a subset of a given set A according to any predicate ϕ. The formed subset is unique by the axiom of extensionality, and thus given a notation:

$$\{x \in A \mid \phi\}.$$

Axiom schema of replacement

For any formula $\phi(\overline{z}, A, x, y)$, the formula

$$\forall \overline{z},\ \forall A,\ (\forall x \in A,\ \exists! y,\ \phi) \Rightarrow \exists B,\ \forall x \in A,\ \exists y \in B,\ \phi$$

is an axiom, where $\exists! w,\ \psi$ is the abbreviation for $\exists w,\ \psi \wedge (\forall v,\ \psi[w := v] \Rightarrow v = w)$.

This axiom schema literally says that if ϕ is a functional (many-one) relation, it has an image as a set. It is used to regard functions as sets, enumerate sets, or define recursive functions.

If $\phi(\overline{z}, A, x, y)$ is a functional relation $y = F(x)$, a notation is given (by extensionality and separation):

$$\{F(x) \mid x \in A\}.$$

Axiom of pairing

The formula

$$\forall x,\ \forall y,\ \exists z,\ x \in z \wedge y \in z$$

is an axiom. The corresponding notation (using separation and extensionality) is:

$$\{x, y\},$$

which denotes the set which exactly contains x and y as elements.

The notation for singleton set is also defined:

$$\{x\} = \{x, x\}.$$

Axiom of union

The formula

$$\forall X,\ \exists y,\ \forall x,\ x \in X \Rightarrow x \subseteq y$$

is an axiom, where $x \subseteq y$ is a notation for the formula $\forall z,\ z \in x \Rightarrow z \in y$. The corresponding notation (using separation and extensionality) is:

$$\bigcup X,$$

which denotes the set such that $\forall w,\ (w \in \bigcup X \iff \exists x,\ w \in x \wedge x \in X)$ holds.

There are also derived notations.

$$x \cup y = \bigcup \{x, y\},$$

$$\bigcup_{x \in X} F(x) = \bigcup \{F(x) \mid x \in X\}.$$

Axiom of powerset

The formula

$$\forall x,\ \exists Y,\ \forall y,\ y \subseteq x \Rightarrow y \in Y$$

is an axiom. The corresponding notation (using separation and extensionality) is:

$$\mathscr{P}(x),$$

which denotes the set such that $\forall y,\ (y \in \mathscr{P}(x) \iff y \subseteq x)$ holds.

Axiom of infinity

The formula

$$\exists x, \emptyset \in x \wedge \forall y,\ y \in x \Rightarrow S(y) \in x$$

is an axiom, where $\emptyset$ is a notation for the set such that $\forall w,\ w \notin \emptyset$ holds (by separation and extensionality), and $S(x) = x \cup \{x\}$.

The notation ω denotes the set such that $\forall X,\ (\emptyset \in X \wedge \forall y,\ y \in X \Rightarrow S(y) \in X) \Rightarrow \omega \subseteq X$ holds. Its existence and uniqueness follow from the axioms of infinity, separation, and extensionality.

Axiom of regularity

The formula

$$\forall x,\ \exists y,\ y \in x \wedge \forall z,\ z \in x \Rightarrow z \notin y$$

is an axiom.

Unlike other axioms, this axiom does not provide an explicit discipline about how to construct a new set form given ones. Instead, it states that $\in$ relation is well-founded everywhere in the universe of sets, thus globally enabling one to perform induction on $\in$.

Miscellaneous notations

So far the enumeration of the axioms of ZF has been completed. We complement them with some frequently used notations and encodings:

Notation for (Kuratowski) ordered pairs: $\langle x, y\rangle = \{x, \{x, y\}\}$.

Notation for cartesian products: For any set X and Y, the notation $X \times Y$ denotes the set such that $\forall w,\ (w \in X \times Y \iff \exists x,\ \exists y,\ x \in X \wedge y \in Y \wedge w = \langle x, y\rangle)$ holds. Either powerset or replacement is used to construct $X \times Y$.

Encoding of functions Functions in set theory are realization of functional (many-one) relation as sets. Usually a function is defined to be a set f such that

$$\exists x,\ \exists y,\ (f \subseteq x \times y) \wedge (\forall z \in x, \exists! w, \langle z, w\rangle \in f).$$

Given a function f, its domain $\mathrm{dom}(f)$ and range $\mathrm{ran}(f)$ are defined as follows:

$$\mathrm{dom}(f) = \left\{ x \in \bigcup\bigcup f \;\middle|\; \exists y,\ \langle x, y\rangle \in f \right\},$$

$$\mathrm{ran}(f) = \left\{ y \in \bigcup\bigcup f \;\middle|\; \exists x,\ \langle x, y\rangle \in f \right\}$$

Axiom of choice

In addition to ZF, we also formulate additional axioms. The first one is the axiom of choice, which is quite useful if not necessary in mathematics, especially in our later formalization of convex sets.

We define the axiom of choice in terms of choice functions. A choice function for set A is a function f such that $\mathrm{dom}(f) = \mathscr{P}(A) \setminus \emptyset$ and for any nonempty $S \subseteq A$ and any $a \in S$, $f(a) \in S$ holds.

The axiom of choice is a statement that every set has a choice function. We are going to seriously deal with the axiom of choice and choice functions in Chapter 5.

2.4 Type Theory

We turn our attention to the formalized part of our work, which is based on the Calculus of Inductive Constructions, a variant of type theory.

Type theory is the common name for many different formal theories of collections. In type theories, collections are called *types*. Individuals in types are called *elements* as in set theory, or *terms* when respecting the formal nature of type theory. When an element t belongs to a type T, we say "t has type T" and denote it by $t : T$.

The defining property of type theories is that different types cannot be mixed for free. For example, for elements $a : A$ and $b : B$ of two different types and a function $f : A \to C$ whose domain is A, the expression $f(a)$ makes sense and has type C while $f(b)$ is nonsensical and does not have a type. One cannot even compare the elements of two different types. Such a comparison is always available in ZF since the primitive equality is present in first-order logic. Often two sets in ZF are indeed identified by the axiom of extensionality, which is neither present in general in type theory. One is also prohibited to naively form a collection like $\{a, b\}$, which cannot be a type without specific axioms justifying the construction. These examples are contrary to the usual habit in set theory, where any function can take any element of any set as input and the constructions like union are allowed for any sets.

Well then, what is the point of using such a restrictive foundation instead of set theory? The answer is safety. Such mixture of collections as mentioned in the above examples happens most often when a mathematician is confused about what collections are being manipulated. Type-theoretic distinction is actually not a restriction. It is rather like a sketch that helps a mathematician to organize the details of proofs without confusions.

2.4.1 Calculus of constructions

The Calculus of Inductive Constructions [Paulin-Mohring, 2015] is based on its core language, the Calculus of Constructions (CC) [Coquand and Huet, 1988].

CC is not based on first-order logic. Rather it directly axiomatizes what types and terms (elements) are. Here is the syntax for terms of CC.

$$\begin{array}{lllrl} \texttt{terms} & \ni & t & ::= & x \quad (x \in \texttt{variables}) \\ & & & | & s \quad (s \in \texttt{sorts}) \\ & & & | & (\lambda x : t.t) \quad (x \in \texttt{variables}) \\ & & & | & (tt) \\ & & & | & (\Pi x : t.t) \quad (x \in \texttt{variables}). \end{array}$$

CC is based on Church's lambda calculus [Church, 1932, Church, 1936], where the basic objects are functions, not sets. In the above definition, $(\lambda x : t_1.t_2)$ denotes the construction of a function, which is $((x \in t_1) \mapsto t_2)$ in an ordinary notation. $(t_1 t_2)$ is a function application $t_1(t_2)$.

$(\Pi x : t_1.t_2)$ is a product type whose elements are $(\lambda x : t_3.t_4)$-s. If the variable x does not occur in t_2, $(\Pi x : t_1.t_2)$ is abbreviated as $t_1 \to t_2$. In CC, types are special terms, and types also have their types. The elements of `sorts` are called *sorts*, which are intuitively large types of all smaller types.[5]

We want to discuss what the type of a given term is. If variables occur in the term, they must be given types beforehand by a *context*. A context is a sequence of pairs of a variable and a type, denoting the assignment of types to variables.

$$\begin{array}{lllrl} \texttt{contexts} & \ni & \Gamma & ::= & nil \\ & & & | & \Gamma; x : t \quad (x \in \texttt{variables}) \end{array}$$

A ternary relation consisting of a context, a term, and a type $\Gamma \vdash t : T$ is called a *typing judgment* or just *judgment*. The axioms of CC is given in a way that they generate judgments based on other judgments. We follow [Barendregt, 1991] for the presentation of the axioms.

(sort axiom)

$$\frac{}{nil \vdash \texttt{Prop} : \texttt{Type}}$$

[5] A sort in CC does not contain itself. Otherwise, it leads to a logical inconsistency [Girard, 1972].

(start)

$$\frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : T}, (x \notin \Gamma)$$

(weakening)

$$\frac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B}, (x \notin \Gamma)$$

(abstraction)

$$\frac{\Gamma, x : A \vdash b : B \quad \Gamma \vdash (\Pi x : A.B) : s}{\Gamma \vdash (\lambda x : A.b) : (\Pi x : A.B)}$$

(application)

$$\frac{\Gamma \vdash F : (\Pi x : A.B) \quad \Gamma \vdash a : A}{\Gamma \vdash Fa : B[x := a]}$$

(product)

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash (\Pi x : A.B) : s_2}, (s_1, s_2 \in \texttt{sorts})$$

(conversion)

$$\frac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s}{\Gamma \vdash A : B'}, (B =_\beta B')$$

These fraction-like notations are to be read as (metatheoretic) implication rules: the formulas above the bar are assumptions and the formula below the bar is the conclusion.

- The sort-axiom rule says that the sort `Prop` is an element of the sort `Type`.
- The weakening rule is structural, and says that a context can be supplied with extra typing information, if the type properly has a sort.
- The abstraction rule says that the elements of a product type $\Pi x : A.B$ are functions $\lambda x : A.b$, and such a function exists when its body $b : B$ is given under the assumption on the variable to be abstracted $x : A$. The resulting type must also properly have a sort ($(\Pi x : A.B) : s$).
- The application rule says for a function application Fa, the resulting type $B[x := a]$ changes depending on the argument a.
- The product rule says that the type $\Pi x : A.B$ has a sort if A and B have sorts, and the resulting type has the same sort as B. In particular, large products such as the product of all types in the sort `Prop`, is again in `Prop`:

$$\frac{\vdash \texttt{Prop} : \texttt{Type} \quad x : \texttt{Prop} \vdash x : \texttt{Prop}}{\vdash (\Pi x : \texttt{Prop}.x) : \texttt{Prop}}$$

In the last product rule, we can define a (large) product in `Prop` by referring to the entirety of `Prop` itself. This property is called the *impredicativity*, and `Prop` is called the *impredicative sort*.

2.4.2 Calculus of inductive constructions

The Calculus of Inductive Constructions (CIC) extends CC with two features, namely user-definable inductive types, and a countable hierarchy of $\texttt{Type}_i, i \in \mathbb{N}$ universes.

The definability of inductive types are motivated by inductively defined sets which are abundant in ordinary mathematics [Aczel, 1977]. A typical example is the set of natural numbers.

The countable hierarchy of types follows the Martin-Löf intuitionistic type theory [Martin-Löf, 1984], which enables the formalization of theories that involve large sets such as category theory.

The precise definition of CIC is much larger than CC and has many variations. Therefore, we do not rather formulate the whole CIC, but grasp it by the semantical features and examples we actually utilize.

2.5 Set-theoretic semantics

We have so far introduced the set theory ZF for the informal mathematics and the type theory CIC for the formalized mathematics. In order to connect these two, enabling the type theory to work as a tool to reason about informal mathematics, we have to connect these two theories. The glue that binds the informal mathematics and the formalization is the interpretation (semantics) of type theory into set theory.

There have been many variants of set-theoretic semantics in the literature, according to the slight variations on the versions of CIC or the different purposes.

- There is an HF-model and IZF-model of CC [Barras, 2010].
- A fragment of CIC is proved to be equiconsistent with ZF plus countably many inaccessible cardinals [Werner, 1997].
- By Reynold's result [Reynolds, 1984], there is no non-trivial set-theoretic model of the impredicative sort. Therefore if we want to build a set-theoretic semantics of CC, the semantics of the impredicative sort have to be dealt with separately. Due to this issue, the handling of `Prop` in the set-theoretic semantics (proof-irrelevant semantics) is subtle, and actually pretty tricky when proving the soundness property with respect to β-conversions [Miquel and Werner, 2003].
- The set-theoretic semantics that incorporates the cumulative hierarchy of sorts (the subsumption rule) was presented by Lee and Werner [Lee and Werner, 2011].
- As shown in Girard's lecture notes, inductive definitions in the impredicative sort are interpreted by the Church encoding [Girard et al., 1989]. On the other hand, inductive definitions

in the predicative sorts are interpreted as inductive definitions in the universe of sets [Aczel, 1977].

2.5.1 Grothendieck universe

In order to translate the type-theoretic universes to set theory, we need similar large sets, namely Grothendieck universes.

A set $\mathcal{U}$ is called a Grothendieck universe if the following conditions are met.

- $\emptyset \in \mathcal{U}$.
- $\mathcal{U}$ is transitive:
$$\forall x \in \mathcal{U},\ x \subseteq \mathcal{U}.$$
- $\mathcal{U}$ is closed under the powerset operation:
$$\forall x \in \mathcal{U},\ \mathscr{P}(x) \in \mathcal{U}.$$
- $\mathcal{U}$ is closed under the indexed union operation: for any $I \in \mathcal{U}$ and function f such that $\forall i \in I,\ f(i) \in \mathcal{U}$,
$$\bigcup_{i \in I} f(i) \in \mathcal{U}.$$

Every Grothendieck universe is known to be equal to the κ-th von Neumann universe for some inaccessible cardinal κ [Williams, 1969], which validates every axiom of ZF relativized to it, thus forming a set-model of ZF. This in turn means that, by Gödel's second incompleteness theorem, the existence of Grothendieck universes cannot be proved inside ZF. We can only add them through an axiom.

We assume an axiom that there is an $\mathbb{N}$-indexed increasing sequence of inaccessible cardinals, or equivalently, the existence of countably many Grothendieck universes increasing along $\mathbb{N}$. Note that in this setting, every Grothendieck universe $\mathcal{U}$ is an element of some larger universe $\mathcal{U}^+$.

This corresponds to the $\texttt{Type}_i$ hierarchy of Coq type universes which is also indexed by natural numbers $i \in \mathbb{N}$.

2.6 Formalization techniques in Coq

We are going to use Coq as a tool to apply the so far presented foundational settings to actual formalization problems. In order to make the resulting code clean and easily reusable, we employ several techniques specific to Coq .

2.6.1 Type inference and canonical structures

In informal mathematics, sets are transparently included in their superset. For example, elements of the set of positive reals $\mathbb{R}_{>0}$ are also elements of $\mathbb{R}$ without any modification. $\mathbb{R}_{>0}$ also inherits algebraic operations from $\mathbb{R}$. An addition of positive reals is again a positive real. A subtraction may not be positive, but still fit in $\mathbb{R}$.

However, in the formalization, $\mathbb{R}$ and $\mathbb{R}_{>0}$ are represented by different types, say `R` and `Rpos` respectively. As we explained in Section 2.4, different types cannot be naively mixed. That is, elements of `Rpos` do not belong `R`, nor a function defined on `R` can be simply applied to elements of `Rpos`.

Coq has several mechanisms to alleviate this inconvenience. Among them, we rely on *canonical structures* [Mahboubi and Tassi, 2013].

2.6.2 Packed classes

In order to formalize various theories and their extensions, we employ a style of definition in Coq called *packed classes* [Garillot et al., 2009]. We explain it by an example. Suppose that we want to define the theory of groups and then extend it to ordered groups.

The signature for groups is like this:

- Carrier set X.
- Unit $e \in X$.
- Inverse $\iota : X \to X$.
- Multiplication $_ * _ : X \times X \to X$.
- Associativity law: $x * (y * z) = (x * y) * z$.
- Unit law: $x * e = e * x = x$.
- Inverse law: $x * \iota(x) = \iota(x) * x = e$.

We are going to turn the definition into Coq code that represents the collection of all groups. A packed class consists of basically three parts of code: *mixin*, *class*, and *structure*. The first step is to turn the signature into a mixin:

```
Record mixin_of (X : Type) : Type := Mixin {
  e : X;
  inv : X -> X;
  mul : X -> X -> X where "x * y" := (mul x y);
  _ : forall x y z, x * (y * z) = (x * y) * z;
```

```
  _ : forall x, x * e = x;
  _ : forall x, e * x = x;
  _ : forall x, x * (inv x) = e;
  _ : forall x, (inv x) * x = e }.
```

As seen in this example, a mixin is just a signature with its base structure (the carrier set X in this case) picked as its key parameter (`(X : Type)` in the code).

The structure of groups is then obtained by combining some set X with the the mixin instantiated by X:

```
Structure t : Type :=  Pack {X : Type ; class : mixin_of X}.
```

These `Record mixin_of` and `Structure t` are given quite universal names. These names are contained in a `Module` of Coq, which hides these names from the public environment. We do not further dig into the details of `Module` system, as we only need to know that duplicate names are nicely managed and not harmful.

The names we use in the public environment are to be explicitly given as follows:

```
Notation grp := t.
```

By this declaration of notation, we can refer to the structure `t` above (which is hidden in a `Module`) by the public name `grp`.

Next, let us define an ordered group to be a group with a partial ordering which is coherent with the multiplication:

- Group G.
- Ordering $(_ \leq _) \subseteq G \times G$.
- Reflexivity law: $x \leq x$.
- Transitivity law: $x \leq y \wedge y \leq z \Rightarrow x \leq z$.
- Antisymmetry law: $x \leq y \wedge y \leq x \Rightarrow x = y$.
- Right coherence law: $x \leq y \Rightarrow x * z \leq y * z$.
- Left coherence law: $x \leq y \Rightarrow z * x \leq z * y$.

The corresponding mixin looks as follows:

```
Record mixin_of (G : grp) : Type := Mixin {
  leq : G -> G -> Prop where "x <= y" := (leq x y);
  _ : forall x, x <= x;
  _ : forall y x z, x <= y -> y <= z -> x <= z;
```

```
  _ : forall x y, x <= y /\ y <= x -> x = y;
  _ : forall x y z, x <= y -> x * z <= y * z;
  _ : forall x y z, x <= y -> z * x <= z * y }.
```

Now an ordered group is a mixture of group and ordering structures. A class is such a mixture of mixins:

```
Record class_of (X : Type) := Class {
  base : Group.mixin_of X;
  mixin : mixin_of (Group.Pack base) }.
```

Note that while the mixin of ordered group takes a group G as the key parameter, the class takes an unstructured carrier set X. Then the structure of ordered groups is a combination of a carrier set X and the class instantiated by X:

```
Structure t : Type := Pack {X : Type; class : class_of X}.
```

Chapter 3

Formalization of Convex Spaces

3.1 Introduction

The notion of convex sets appears in various mathematical theories. A subset X of a real vector space is called a convex set if, for any $x, y \in X$ and $p \in [0,1]$, their *convex combination* $px+(1-p)y$ is again in X. One basic use of it is to define the convexity of functions. A function f is said to be convex if $f(px + (1-p)y) \leq pf(x) + (1-p)f(y)$ for any convex combination $px + (1-p)y$. Thus, convex sets are natural domains for convex functions to be defined on. Good examples of these notions can be found in information theory, where convexity is a fundamental property of important functions such as logarithm, entropy, and mutual information. Our InfoTheo library has a formalization of textbook proofs [Cover and Thomas, 2006] of such results.

In the course of formalizing such convexity results, we find that axiomatizing convex sets is a useful step which provides clarity and organizability in the results. We abstract the usual treatment of convex sets as subsets of some vector space and employ an algebraic theory of *convex spaces*, which was introduced by Stone [Stone, 1949]. The formalization uses the packed class construction (Section 2.6.2), so as to obtain generic notations and lemmas, and more importantly, to be able to combine structures.

We also formalize an embedding of convex spaces into *conical spaces* (a.k.a. cones or real cones [Varacca and Winskel, 2006]), which we find an indispensable tool to formalize convex spaces. Examples in the literature avoid proving properties of convex spaces directly and choose to work in conical spaces. This is especially the case when their goal can be achieved either way [Kirch, 1993, Varacca and Winskel, 2006]. Some authors suggest or show that the results in conical spaces can be backported to convex spaces [Flood, 1981, Keimel and Plotkin, 2016]. We apply this method to enable additive handling of convex combination. By formalizing the relationship between convex and conical spaces, we work out short proofs of a number of lemmas on convex spaces. Among

them is Stone's key lemma [Stone, 1949, Lemma 2], whose proof is often omitted in the literature despite its fundamental role in the study of convex spaces.

Another aspect of convex spaces is the relationship to probabilistic distributions. From any set, one can freely generate a convex space by formally taking all convex combinations among the elements of the set. The resulting convex space can be seen as a set of distributions over the original set, since the formal convex combinations are equivalent to distributions over the given points. By this construction, convex spaces serve as a foundation for the algebraic and category-theoretic treatments of probability. This allows for another application in the semantics of probabilistic and nondeterministic programming [Jacobs, 2010, van Heerdt et al., 2018]. We show how elementary definitions in such semantics are written using convex spaces. Category-theoretic developments over them are given in Chapters 4 and 5.

3.2 Convex spaces

Let us begin with the definition of convex spaces. As mentioned in the introduction, convex spaces are an axiomatization of the usual notion of convex sets in vector spaces. It has a long history of repeated reintroduction by many authors, often with minor differences and different names: barycentric calculus [Stone, 1949], semiconvex algebra [Świrszcz, 1974], or, just, convex sets [Jacobs, 2010].

We define convex spaces following Fritz [Fritz, 2009, Definition 3.1].

Definition 3.2.1 (`Module ConvexSpace` in [Infotheo, 2020, `probability/convex_choice.v`]). A convex space is a structure for the following signature:

- Carrier set X.
- Convex combination operations $(_ \triangleleft p \triangleright _) : X \times X \to X$ indexed by $p \in [0, 1]$.
- Unit law: $x \triangleleft 1 \triangleright y = x$.
- Idempotence law: $x \triangleleft p \triangleright x = x$.
- Skewed commutativity law: $x \triangleleft 1 - p \triangleright y = y \triangleleft p \triangleright x$.
- Quasi-associativity law: $x \triangleleft p \triangleright (y \triangleleft q \triangleright z) = (x \triangleleft r \triangleright y) \triangleleft s \triangleright z$, where $s = 1 - (1 - p)(1 - q)$ and $r = \begin{cases} p/s & \text{if } s \neq 0 \\ 0 & \text{otherwise} \end{cases}$. (Note that r is irrelevant to the value of $(x \triangleleft r \triangleright y) \triangleleft s \triangleright z$ if $s = 0$).

□

We can translate this definition to Coq as a *packed class* [Garillot et al., 2009] with the following mixin interface:

```
Record mixin_of (X : Type) : Type := Mixin {
  conv : X -> X -> prob -> X where "x <| p |> y" := (conv x y p);
  _ : forall x y, x <| `Pr 1 |> y = x ;
  _ : forall x p, x <| p |> x = x ;
  _ : forall x y p, x <| p |> y = y <| `Pr p.~ |> x;
  _ : forall (p q : prob) (x y z : X),
      x <| p |> (y <| q |> z) = (x <| [r_of p, q] |> y) <| [s_of p, q] |> z }.
```

There are some notations and definitions to be explained. The `prob` in the above Coq code denotes the closed unit interval $[0, 1]$. `` `Pr r `` is a notation for a real number `r` equipped with a canonical proof that $0 \leq \texttt{r} \leq 1$. `p.~` is a notation for $1 - \texttt{p}$. `[s_of p, q]` is a notation for $1 - (1 - \texttt{p})(1 - \texttt{q})$, and `[r_of p, q]` for `p/[s_of p, q]`.

Intuitively, one can regard the convex combination as a probabilistic choice between two points. At line 3, the left argument is chosen with probability 1. Lines corresponding to idempotence, skewed commutativity, and quasi-associativity follow.

An easy example of convex space is the real line $\mathbb{R}$, whose convex combination is expressed by ordinary addition and multiplication as $pa + (1-p)b$. Probability distributions also form a convex space. In the formalization, the type `fdist A` of distributions over any finite type `A` [Affeldt et al., 2014] is equipped with the convex space structure, with the convex combination of two distributions d_1, d_2 are defined pointwise as $x \mapsto pd_1(x) + (1-p)d_2(x)$. These examples are subsumed in the general case that any convex set in any real vector space is a convex space.

As a result of the packed class construction, we obtain the type `convType` of all types which implicitly carry the above axioms. Then, each example of convex spaces are declared to be canonically a member of `convType`, enabling the implicit inference of the appropriate convex space structure. These two implicit inference mechanisms combined make the statement of generic lemmas on convex spaces simple and applications easy.

3.3 Multiary convex combination

Convex spaces can also be characterized by multiary convex combination operations, which combine finitely many points $x_0, \ldots, x_{n-1}$ at once, according to some finite probability distribution d over the set $\{0, \ldots, n-1\}$. A definition of convex spaces based on multiary operations is given as follows (See for example [Bonchi et al., 2017, Definition 5] and [van Heerdt et al., 2018, Sect. 2.1]).

Definition 3.3.1 (Convex space (multiary version))**.** A convex space based on multiary operations is a structure for the following signature:

- Carrier set X.

- Multiary convex combination operations

$$\begin{array}{ccc} X^n & \to & X \\ (x_i)_{i<n} & \mapsto & \bigoplus_{i<n} d_i x_i \end{array}$$

indexed by natural numbers n and probability distributions d (whose valuation at i being written d_i) over the set $\{0, \ldots, n-1\}$.

- Projection law: if $d_j = 1$, $\bigoplus_{i=0}^{n-1} d_i x_i = x_j$.
- Barycenter law: $\bigoplus_{i=0}^{n-1} d_i (\bigoplus_{j=0}^{m-1} e_{i,j} x_j) = \bigoplus_{j=0}^{m-1} (\sum_{i=0}^{n-1} d_i e_{i,j}) x_j$.

□

This multiary convex structure and the binary one given in Section 3.2 are indeed equivalent, in the sense that the multiary and binary operators interpret each other satisfying the needed axioms, and the interpretations cancel out when composed [Infotheo, 2020, `Module AltConvexSpaceEquiv`]. While the binary axiomatization is easier to instantiate, the multiary version exhibits the relationship to probability distributions. Therefore we want to establish this equivalence before further working on other constructions over convex spaces.

The first step of showing the equivalence is to define the multiary operator in terms of the binary one.

Definition 3.3.2 (`Convn` in [Infotheo, 2020, `probability/convex_choice.v`])**.** Let I_n denote the set $\{0, \ldots, n-1\}$, $d : \mathrm{I}_n \to [0,1]$ be a finite distribution over I_n, and $x : \mathrm{I}_n \to X$ be a sequence of points in a convex space X. Then the multiary convex combination of these points and distribution is denoted by $\bigoplus_{i=0}^{n-1} d_i x_i$ and defined by recursion on n as follows:

$$\bigoplus_{i=0}^{n-1} d_i x_i = \begin{cases} x_0 & \text{if } d_0 = 1 \text{ or } n = 1 \\ x_0 \triangleleft d_0 \triangleright (\bigoplus_{i=0}^{n-2} d'_i x_{i+1}) & \text{otherwise} \end{cases}$$

where d' is a new distribution: $d'_i = d_{i+1}/(1-d_0)$.

□

Note that in our actual Coq code, $\bigoplus_{i=0}^{n-1} d_i x_i$ appears in forms `\Conv_d x` or `Convn d x`, indicating more explicitly that the operation takes two arguments d and x.

In order to establish the desired equivalence, we have technical difficulties in proving the properties of multiary operator based on the above recursive definition. In the literature, it is justified by referring to (though omitting the detail of) the seminal article by Stone [Stone, 1949] (see, e.g., [Jacobs, 2010, Theorem 4], [Bonchi et al., 2017, Proposition 7]). We are going to see in the next section that the original proof by Stone is better formalized by transporting the argument to conical spaces.

3.4 Conical spaces and embedded convex spaces

The definition of multiary convex combination operator in the previous section relied on recursion. This makes the definition look complicated, and moreover, the algebraic properties of the combination difficult to see. If we consider the special case of convex sets in a vector space, the meaning of multiary combinations and the algebraic properties become evident:

$$\bigoplus_{i=0}^{n-1} d_i x_i = d_0 x_0 + \cdots + d_{n-1} x_{n-1}.$$

The additions on the right-hand side are of vectors, and thus are associative and commutative. This means that the multiary combination on the left-hand side is invariant under permutations or partitions on indices. We want to show that these invariance properties are also satisfied generally in any convex space.

However, the search for the proofs is painful if naively done. This is because binary convex combination operations satisfy associativity and commutativity only through cumbersome parameter computations. For example, a direct proof of the permutation case involves manipulations on the set of indices n $(= \{0, \ldots, n-1\})$ and on the symmetry groups, which is a fairly long piece of combinatorics [Stone, 1949, Lemma 2].

We present a solution to this complexity by transporting the arguments on convex spaces to a closely related construction of conical spaces. Conical spaces are an abstraction of cones in real vector spaces just like convex spaces are an abstraction of convex sets. Like convex spaces, the definition of conical spaces appears in many articles. We refer to the ones by Flood (called semicone there) [Flood, 1981] and, by Varacca and Winskel (called real cone there) [Varacca and Winskel, 2006]:

Definition 3.4.1 (Conical space)**.** A conical space is a semimodule over the semiring of non-negative reals. That is, it is a structure for the following signature:

- Carrier set X.
- Zero $\mathbf{0} : X$.
- Addition operation $_ + _ : X \times X \to X$.
- Scaling operations $c_ : X \to X$ indexed by $c \in \mathbb{R}_{\geq 0}$.
- Associativity law for addition: $x + (y + z) = (x + y) + z$.
- Commutativity law for addition: $x + y = y + x$.
- Associativity law for scaling: $c(dx) = (cd)x$.

- Left-distributivity law: $(c + d)x = cx + dx$.
- Right-distributivity law: $c(x + y) = cx + cy$.
- Zero law for addition: $\mathbf{0} + x = x$.
- Left zero law for scaling: $0x = \mathbf{0}$.
- Right zero law for scaling: $c\mathbf{0} = \mathbf{0}$.
- One law for scaling: $1x = x$.

□

We display this definition only to show that conical spaces have straightforward associativity and commutativity. The formalization is instead elaborated on the embedding of convex spaces into canonically constructed conical spaces, which appeared in the article by Flood [Flood, 1981]. We build on top of each convex space X, the conical space S_X of its "scaled points".

Definition 3.4.2 (`scaled_pt`, `addpt`, and `scalept` in [Infotheo, 2020, `probability/convex_choice.v`])**.** Let X be a convex space. We define a set S_X which becomes a conical space with the following addition and scaling operations.

$$S_X := (\mathbb{R}_{>0} \times X) \cup \{\mathbf{0}\}.$$

That is, the points of S_X are either a pair (p, x) of $p \in \mathbb{R}_{>0}$ and $x \in X$, or a new additive unit $\mathbf{0}$. Addition of points $a, b \in S_X$ is defined by cases to deal with $\mathbf{0}$:

$$a + b := \begin{cases} (r + q, x \triangleleft r/(r + q) \triangleright y) & \text{if } a = (r, x) \text{ and } b = (q, y) \\ a & \text{if } b = \mathbf{0} \\ b & \text{if } a = \mathbf{0} \end{cases}.$$

Scaling $a \in S_X$ by $p \in \mathbb{R}_{\geq 0}$ is also defined by cases:

$$pa := \begin{cases} (pq, x) & \text{if } p > 0 \text{ and } a = (p, x) \\ \mathbf{0} & \text{otherwise} \end{cases}.$$

□

We omit here the proofs that S_X with these addition and scaling satisfies the conical laws. They are proved formally in [Infotheo, 2020, `probability/ convex_choice.v`] (See the Coq scripts `addptC`, `addptA`, `scalept_addpt`, etc.).

Properties of the underlying convex spaces are transported into and back from this conical space, through an embedding:

Definition 3.4.3 (`S1` in [Infotheo, 2020, `probability/convex_choice.v`]).

$$\begin{array}{rccc} S_1 : & X & \rightarrowtail & S_X \\ & x & \mapsto & (1, x) \end{array}$$

□

Convex combinations in X are mapped by S_1 to additions in S_X.

Lemma 3.4.4 (`S1_convn` in [Infotheo, 2020, `probability/convex_choice.v`]).

$$S_1(\bigoplus_{i=0}^{n-1} d_i x_i) = \sum_{i=0}^{n-1} d_i S_1(x_i).$$

□

The right-hand side of the lemma is a conical sum, which behaves like an ordinary linear sum thanks to the conical laws, and enjoys a good support from the big operator library of MathComp [Bertot et al., 2008].

With these preparations, properties such as [Stone, 1949, Lemma 2] can be proved in a few lines of Coq code:

Lemma 3.4.5 (`Convn_perm` in [Infotheo, 2020, `probability/convex_choice.v`]).

$$\bigoplus_{i=0}^{n-1} d_i x_i = \bigoplus_{i=0}^{n-1} (d \circ s)_i (x \circ s)_i,$$

where s is any permutation on the set of indices n. □

The proof of the barycenter property [Stone, 1949, Lemma 4] from Section 3.3 is based on the same technique (see `Convn_convnfdist` in [Infotheo, 2020, `probability/convex_choice.v`]).

A way to understand this conical approach is to start from Stone's definition of convex spaces [Stone, 1949]. He uses a quaternary convex operator $(x, y; \alpha, \beta)$ where x and y are points of the space, and α and β are non-negative coefficients such that $\alpha + \beta > 0$. The values of this operator are quotiented by an axiom to be invariant under scaling, removing the need to normalize coefficients for associativity. This amounts to regarding a convex space as the projective space of some conical space.

The definition of S_X is a concrete reconstruction of such a conical space from a given convex space X. The benefit of this method over Stone's is the removal of quotients by moving the coefficients from operations to values. We can then use the linear-algebraic properties of conical sums such as the neutrality of zeroes, which had to be specially handled in Stone's proofs (e.g., [Stone, 1949, Lemma 2]).

3.5 Formalization of convex functions

In order to present several applications of convex spaces in the later sections, we define here ordered convex spaces and convex functions.

Ordered convex space is a combined structure of convex space and partial order.

Definition 3.5.1 (`Module OrderedConvexSpace` in [Infotheo, 2020, `probability/convex_choice.v`])**.** An ordered convex space is a structure for the signature extending that of convex spaces:

- Convex space X.
- Ordering relation $(_ \leq _) \subseteq X \times X$.
- Reflexivity law: $x \leq x$.
- Transitivity law: $x \leq y \wedge y \leq z \Rightarrow x \leq z$.
- Antisymmetry law: $x \leq y \wedge y \leq x \Rightarrow x = y$.

□

This signature is translated to a packed class with the following mixin interface:

```
Record mixin_of (T : convType) : Type := Mixin {
  leconv : T -> T -> Prop where "a <= b" := (leconv a b);
  _ : forall a, a <= a;
  _ : forall b a c, a <= b -> b <= c -> a <= c;
  _ : forall a b, a = b <-> a <= b /\ b <= a }.
```

The above definition does not force any interaction between convexity and ordering. It would also be a natural design to include an axiom stating that convex combinations preserve ordering [Keimel and Plotkin, 2016, Section. 2]. We however do not need such interactions for the definition of convex functions, which is our purpose here.

Convexity of a function is defined if its codomain is an ordered convex space. Let A be a convex space and B an ordered convex space until the end of this subsection.

Definition 3.5.2 (`convex_function` in [Infotheo, 2020, `probability/convex_choice.v`])**.** A function $f : A \to B$ is *convex* if, for any $p \in [0, 1]$ and $x, y \in A$, $f(x \triangleleft p \triangleright y) \leq f(x) \triangleleft p \triangleright f(y)$ holds. □

We often want the convexity of functions not over a whole convex space, but some subset of convex space. Such a subset must be a convex set in the sense that it is closed under convex combinations.

Definition 3.5.3 (`is_convex_set` in [Infotheo, 2020, `probability/convex_choice.v`])**.** A subset D of convex space is a *convex set* if, for any $p \in [0, 1]$ and $x, y \in D$, $x \triangleleft p \triangleright y \in D$. □

Definition 3.5.4 (`convex_function_in` in [Infotheo, 2020, `probability/convex_choice.v`]). Let D be a convex set in A. A function $f : A \to B$ is *convex in* D if, for any $p \in [0,1]$ and $x, y \in D$, $f(x \triangleleft p \triangleright y) \leq f(x) \triangleleft p \triangleright f(y)$ holds. □

Concave functions are defined similarly by reversing the ordering in the definition of convex functions (`concave_function` and `concave_function_in` in [Infotheo, 2020, `probability/convex_choice.v`]). When the codomain of a function f is $\mathbb{R}$, the prototypical example of an ordered convex space, it is also easy to prove that f is concave if $-f$ is convex (with the ring structure for functions being defined pointwise).

3.6 Applications in information theory

Application 1: concavity of log

As a first application, we prove that the real logarithm function is concave. The concavity of logarithm is frequently utilized in information theory, for example, properties of data compression depend on it [Affeldt et al., 2018].

The definition of logarithm we employ in Coq has the entire $\mathbb{R}$ as its domain by setting $\log(x) = 0$ for $x \leq 0$. The statement of concavity is then relativized back to the subset $\mathbb{R}_{>0}$. [1]

Lemma 3.6.1 (`log_concave` in [Infotheo, 2020, `probability/ln_facts.v`]). *The extended logarithm function*

$$x \mapsto \begin{cases} \log(x) & \text{if } x \in \mathbb{R}_{>0} \\ 0 & \text{otherwise} \end{cases}$$

is concave in $\mathbb{R}_{>0}$. □

We also show the corresponding line of statement in the Coq file for the sake of later comparison:

```
Lemma log_concave : concave_function_in Rpos_interval log.
```

Besides the use of convex spaces, the heart of the proof is the fact that a function whose second derivative is non-negative is convex (`Section twice_derivable_convex` in [Infotheo, 2020, `probability/convex_choice.v`]). The proof proceeds using the Coq formalization of real analysis, using our formalization of convex spaces as an additional abstraction layer of convexity.

[1] Note that this manner of restricting the domain of functions rather in their properties than in the definitions is a design choice often found in Coq. This choice makes it possible for functions such as logarithm to be composable without being careful of their domains and ranges, and leads to a clean separation between definitions and properties of functions in the formalization.

Application 2: concavity of entropy

We go on to formalize the convexity (or concavity) proofs of other information-theoretic functions. A typical one is Shannon's entropy:

Definition 3.6.2 (`entropy` in [Infotheo, 2020, `probability/convex_choice.v`]). For a probability distribution P over a finite set A, its entropy $H(P)$ is

$$H(P) = -\sum_{a \in A} P(a) \log(P(a)).$$

□

Lemma 3.6.3 (`entropy_concave` in [Infotheo, 2020, `information_theory/convex_fdist.v`]). *H is concave.* □

The entropy is a real-valued function defined on the set of distributions. In the formalization, the type of distributions `fdist A` is endowed with a convex space structure (Section 3.2), and is compatible with our generic definition of convex function. As a result, the line of statement becomes as simple as the `log_concave` above:

```
Lemma entropy_concave : concave_function (fun P : fdist_convType A => `H P).
```

The expression `(fun P : fdist_convType A => `H P)` stands for $(P \mapsto H(P))$, with the explicit mention that `fdist A` is a convex space.

The concavity in this lemma says that entropy increases with averaging, and ensures the intuition that entropy is a measure of disorder (entropy is maximized for uniform distributions), which is a fundamental property of entropy and useful in proving other lemmas.

Application 3: concavity of mutual information

Another example of an information-theoretic function is the mutual information function. In order to define it, we first need the relative entropy function.

Definition 3.6.4 (`div` in [Infotheo, 2020, `probability/divergence.v`]). For probability distributions P and Q over a finite set A, their *relative entropy* $D(P||Q)$ is

$$D(P||Q) = \sum_{a \in A} P(a) \log\left(\frac{P(a)}{Q(a)}\right).$$

□

Then the mutual information is defined as follows.

Definition 3.6.5 (`MutualInfo.mi` in [Infotheo, 2020, `information_theory/chap2.v`]). For a (joint) probability distribution R over a product of finite sets $A \times B$, its mutual information $I(R)$ is

$$I(R) = D(R||P \times Q),$$

where P and Q are the first and second marginal distributions of R respectively, and $P \times Q$ is their product distribution (`Module`s `Bivar` and `ProdFDist` in [Infotheo, 2020, `probability/fdist.v`])

$$P(a) = \sum_{y \in B} R(a, y),$$

$$Q(b) = \sum_{x \in A} R(x, b),$$

$$(P \times Q)(a, b) = P(a)Q(b).$$

□

The statement of concavity is a bit twisted for mutual information. Note first that the joint distribution R in the above definition is equal to a product of its first marginal distribution P and the family $(Q_a)_{a \in A}$ of conditional distributions with respect to P:

$$R(a, b) = P(a)Q_a(b),\ Q_a(b) = R(a, b)/P(a).$$

Observing this in the other direction, any joint distribution R is constructed from a distribution P over A and a family $Q = (Q_a)_{a \in A}$ of distributions over B. These P and Q are independent (functional) variables that determine R, and the concavity of mutual information says that when fixing Q, R is concave with respect to P:

Lemma 3.6.6 (`mutual_information_concave` in [Infotheo, 2020, `information_theory/convex_fdist.v`]). *Let A, B be finite sets and Q be a function from A to the set of distributions over B. Then, the following function from the set of distributions over A to $\mathbb{R}$ is concave:*

$$P \mapsto I(R),$$

where $R(a, b) = P(a)Q_a(b)$. □

As we have seen, this statement involves more complex notions than the previous two applications. Yet the Coq counterpart is again simple, thanks to the development on convex spaces:

```
Lemma mutual_information_concave :
  concave_function (fun P : fdist_convType A => MutualInfo.mi (CJFDist.make_joint P Q)).
```

The proof is formalized according to an informal version present in [Cover and Thomas, 2006, Theorem 2.7.4]. The informal proof involves many manipulations of convex combinations over distributions. It was our original motivation of formalizing convex spaces to abstract those manipulations, enabling a clear and modularized translation of proof steps, which we thought necessary for formalizing the proof.

Aside from the completion of that and similar proofs, other applications in program semantics, which is a totally different domain of research, resulted from the formalization of convex spaces, as seen in the next section and Chapter 5.

3.7 Nonempty convex sets – application to program semantics

We apply our formalization of convex spaces to define the nonempty powerset operation on convex spaces and seek its properties. The constructions and lemmas in this section are later used to define a model of programs involving nondeterminism and probability in Chapter 5.

Hereafter, in this section, we fix a convex space A.

Let us denote the set of nonempty convex sets in A (the nonempty convex powerset of A) by $\mathscr{P}_c(A)$:

Definition 3.7.1 (`necset` in [Infotheo, 2020, `probability/necset.v`]). The convex powerset $\mathscr{P}_c(A)$ of A is

$$\mathscr{P}_c(A) = \{X \subseteq A \mid (X \text{ is a convex set}) \land (X \neq \emptyset)\}$$

□

A nonempty convex powerset again possesses a convex space structure, by lifting the convex combination operator $x \triangleleft p \triangleright y$ to nonempty convex sets:

Definition 3.7.2 (`necset_convType.pre_pre_conv` in [Infotheo, 2020, `probability/necset.v`]). For $p \in [0,1]$ and $X, Y \in \mathscr{P}_c(A)$, their convex combination (the probabilistic choice between X and Y according to p) $X \triangleleft p \triangleright Y$ is

$$X \triangleleft p \triangleright Y = \{x \triangleleft p \triangleright y \mid x \in X \land y \in Y\}$$

□

One then should prove that this lifted operation yields again a nonempty convex set. The corresponding proofs (`necset_convType.pre_pre_conv_convex` and `necset_convType.pre_conv_neq0`) are packed into a nonempty convex set structure (`necset_convType.conv`).

We can define a convex hull of any nonempty set in a convex space:

Definition 3.7.3 (`hull` in [Infotheo, 2020, `probability/convex_choice.v`])**.** For a subset X of A, its hull $\overline{X}$ is

$$\overline{X} = \left\{ \bigoplus_{i=0}^{n-1} d_i x_i \;\middle|\; (n \in \mathbb{N}) \wedge (d \text{ is a distribution over } \{0, \ldots, n-1\}) \wedge (\forall i < n,\ x_i \in X) \right\}$$

□

Hulls are always convex sets (`hull_is_convex` in [Infotheo, 2020, `probability/convex_choice.v`]). In particular, hulls of nonempty sets are nonempty convex sets (`neset_hull_necset` in [Infotheo, 2020, `probability/necset.v`]). This fact leads to the definition of the following operator, which is used to interpret nondeterminism in $\mathscr{P}_c(A)$.

Definition 3.7.4 (`necset_semiCompSemiLattType.pre_op` in [Infotheo, 2020, `probability/necset.v`])**.** For a nonempty $F \subseteq \mathscr{P}_c(A)$, the nondeterministic choice $\bigsqcup F$ is

$$\bigsqcup F = \overline{\bigcup F}$$

□

This operation defines another algebraic theory for nonempty convex powersets, namely that of semicomplete semilattice (Section 5.6). In $\mathscr{P}_c(A)$, the structures of convex space and semicomplete semilattice interact by the following distributivity law: convex combinations distribute over semicomplete semilattice operations.

Lemma 3.7.5 (`necset_semiCompSemiLattConvType.axiom` in [Infotheo, 2020, `probability/necset.v`])**.** *The probabilistic choice distributes over the nondeterministic choice, that is, for $p \in [0,1]$, $X \in \mathscr{P}_c(A)$ and a nonempty $F \subseteq \mathscr{P}_c(A)$,*

$$X \triangleleft p \triangleright \left(\bigsqcup F\right) = \bigsqcup_{Y \in F} (X \triangleleft p \triangleright Y),$$

where $\bigsqcup_{Y \in F} (\ldots Y \ldots)$ is the abbreviation for $\bigsqcup \{\ldots Y \ldots \mid Y \in F\}$. □

The proofs of the above laws are formalized as several lemmas in [Infotheo, 2020, `probability/necset.v`], defining canonical structures for $\mathscr{P}_c(A)$ (`Module`s `necset_convType` and `necset_semiCompSemiLattType`).

We have described $\mathscr{P}_c(A)$ as a place to deal with the semantics of nondeterminism and probability, and their interaction (the *combined choice*). However, in order to utilize the combined choice in computer programs, we want it to be treated as a *computational effect*, which is the theme of Chapter 4. In Chapter 5, we extend the definitions in this section to define the monad for combined choice, which is our solution to the problem of formalizing a model of combined choice.

3.8 Related work

Conical spaces have been known in the literature to work as a nice-behaving replacement of convex spaces when constructing models of nondeterministic computations. Varacca and Winskel [Varacca and Winskel, 2006] used convexity when building a categorical monad combining probability and nondeterminism, but they chose to avoid the problem of equational laws in convex spaces by instead working with conical spaces. There is a similar preference in the study of domain-theoretic semantics of nondeterminism, to a conical structure (d-cones [Kirch, 1993]) over the corresponding convex structure (abstract probabilistic domain [Jones and Plotkin, 1989]). The problem is the same in this case: the difficulty in working with the equational laws of convex spaces. [Keimel and Plotkin, 2009, Tix et al., 2009]

Flood [Flood, 1981] proposed to use conical spaces to investigate the properties of convex spaces. He showed that for any convex space, there is an enveloping conical space and the convex space is embedded in it. (A version of the embedding for convex sets into cones in vector spaces was already present in Semadini's book [Semadini, 1971].) Keimel and Plotkin [Keimel and Plotkin, 2016] extended the idea for their version of ordered convex spaces and applied it in the proof of their key lemma [Keimel and Plotkin, 2016, Lemma 2.8], which is an ordered version of the one proved by Neumann [Neumann, 1970, Lemma 2].

Chapter 4

Formalization of Computational Effects (Equational Reasoning on Monads)

4.1 Introduction

In this chapter, we present a formalization of computational effects in computer programs. When we model a program as a function that maps a set of data to another, the extra parts *not* modeled in the function are called *computational effects*, or simply *effects*. The other functional parts are often said to be *pure*.

Common examples of effects are communications with external devices such as keyboard or monitor (*I/O*), writing and updating the data in memory (*state*), and manipulation of function call stack (*continuation*). These effects are added to programs for efficiency or other practical purposes. I/O is necessary for a program to be meaningful in real applications. State is useful for writing efficient algorithms. Continuation in raw form is a dangerously powerful tool, but useful for implementing other more tame control flow operators.

The goal of this chapter is to provide a framework for writing formal proofs of semantical correctness for programs with effects. In order to build such a framework, we need a mathematical account for effects. Among the many ways to formulate effects, we use monads.

Monad is a category-theoretic device used to augment a category with a possibly infinitary algebraic structure over it [Linton, 1966,Barr and Wells, 1985]. When used for computer programs, monads are applied to data structures and construct extended data structures.

The use of monads for expressing effects was first discovered by Moggi [Moggi, 1989] as an

abstract tool to model effectful computations in a theoretical level. Then it was soon found by Wadler [Wadler, 1990] that monad is realizable as a concrete construct in programming languages for actually writing effectful programs. In both cases, effects are represented by extensions of data structures. Thus the discoveries by Moggi and Wadler give a viewpoint that effects are algebras over data structures.

Plotkin and Power investigated this algebraic aspect of effects and showed that monads defined from algebraic signature are equipped with the corresponding algebraic operations ("generic effects" in [Plotkin and Power, 2001, Plotkin and Power, 2002]). The idea of generic effects is utilized for concrete programs by Gibbons and Hinze [Gibbons and Hinze, 2011]. They extend the theory of monads with various algebraic structures for specific effects. Using these theories, they apply the method of equational reasoning to effects, which is an extension of the method from pure programs to effectful programs.

We present how to formalize a hierarchy of the theories of monads according to [Affeldt et al., 2019], which enables one to formally perform equational reasoning on effectful programs. The target programs are expressed via shallow embedding: they are directly written as Coq functions and interpreted by Coq's semantics. Since we are working in the set-theoretic interpretation of Coq, the data structures used in the programs represent sets, and the monads used for effects are all defined on the category **Set** of sets.

In the following sections, we are going to formalize the theories of functors and monads on **Set**. Placing them at the root of hierarchy, we proceed with extending the theory by several algebraic operators and equational laws corresponding to different effects. We rely on the extensibility of packed classes (Section 2.6.2) to formalize the hierarchy. For each theory, we also provide a model to ensure the consistency of the theory, except for the combined choice monad, whose formalized model is the problem to be solved in Chapter 5.

4.2 Notions from category theory

In this section, we recall several category-theoretic notions until we arrive at the definition of monads. We follow the standard definitions given in the literature, such as Mac Lane's book [Mac Lane, 1998].

First of all, we need the definition of categories. The details may vary among the literature.

Definition 4.2.1 (Category)**.** A category $\mathcal{C}$ is a structure for the following signature:

- Class of objects $\mathcal{C}_0$.
- Classes of morphisms $\hom_{\mathcal{C}}(a, b)$ for $a, b \in \mathcal{C}_0$. The subscript $\mathcal{C}$ will be omitted if obvious. We may also denote by $f : a \to b$ that $f \in \hom(a, b)$.

- Identity morphisms $\mathrm{id}_a : a \to a$ for every $a \in \mathcal{C}_0$. The subscript a will be omitted if obvious.
- Compositions of morphisms $g \circ f : a \to c$ for every $a, b, c \in \mathcal{C}_0$, $f : a \to b$, and $g : b \to c$.
- Unit law: $\mathrm{id}_b \circ f = f \circ \mathrm{id}_a = f$ for any $f : a \to b$.
- Associativity law: $h \circ (g \circ f) = (h \circ g) \circ f$ for any $a, b, c, d \in \mathcal{C}_0$, $f : a \to b$, $g : b \to c$, and $h : c \to d$. □

Let $\mathcal{C}_1$ be the collection of all morphisms: $\mathcal{C}_1 = \bigcup_{a,b \in \mathcal{C}_0} \hom(a, b)$. Categories are classified by the sizes of $\mathcal{C}_0$, $\mathcal{C}_1$, and $\hom(-,-)$s:

- If at least one of $\mathcal{C}_0$ or $\mathcal{C}_1$ is a proper class, the category is said to be *large*.
- If $\hom(a, b)$ is a set for any a and b, the category is *locally small*. $\hom(a, b)$ is then called a *hom-set*.
- If $\mathcal{C}_0$ and $\mathcal{C}_1$ are both sets, the category is *small*.
- If furthermore $\mathcal{C}_0$ and $\mathcal{C}_1$ are elements of a set $\mathcal{U}$, the category is $\mathcal{U}$-*small*.

The definitions of functors and natural transformations are also usual.

Definition 4.2.2 (Functor)**.** Let $\mathcal{C}$ and $\mathcal{D}$ be categories. A functor $F : \mathcal{C} \to \mathcal{D}$ (or just F) from $\mathcal{C}$ to $\mathcal{D}$ is defined by the following signature:

- Object-part function $F_0 : \mathcal{C}_0 \to \mathcal{D}_0$.
- Morphism-part function $F_1 : \mathcal{C}_1 \to \mathcal{D}_1$.
- Graph homomorphism law: If $f \in \hom_{\mathcal{C}}(a, b)$, $F_1(f) \in \hom_{\mathcal{D}}(F_0(a), F_0(b))$.
- Identity law: $F_1(\mathrm{id}_a) = \mathrm{id}_{F_0(a)}$.
- Composition law: $F_1(g \circ f) = F_1(g) \circ F_1(f)$. □

Definition 4.2.3 (Natural transformation)**.** Let $\mathcal{C}, \mathcal{D}$ be categories and $F, G : \mathcal{C} \to \mathcal{D}$ be functors. A natural transformation $\phi : F \rightsquigarrow G$ (or just ϕ) from F to G is defined by the following signature:

- Component morphisms $\phi_a \in \hom_{\mathcal{D}}(F_0(a), G_0(a))$ indexed by all $a \in \mathcal{C}$.
- Naturality law: $\phi_b \circ F_1(f) = G_1(f) \circ \phi_a$ for any $f \in \hom_{\mathcal{C}}(a, b)$ □

Like morphisms, functors and natural transformations are equipped with identities and compositions. An identity functor, which exists on every category, has identity functions as its object and morphism parts. We denote an identity functor on category $\mathcal{C}$ by $\mathrm{Id}_{\mathcal{C}}$, or just Id if $\mathcal{C}$ is obvious. Similarly, there exists for every functor an identity natural transformation whose components are

all identity morphisms. An identity natural transformation on functor F is denoted by Id_F, or just Id if F is obvious.

For functors $F : \mathcal{C} \to \mathcal{D}$ and $G : \mathcal{D} \to \mathcal{E}$, their composition $G \circ F : \mathcal{C} \to \mathcal{E}$ is obtained by composing each parts: $(G \circ F)_i = G_i \circ F_i$ where $i = 0, 1$. For natural transformations, there are two ways of composition. The easier one is vertical composition. Let $F, G, H : \mathcal{C} \to \mathcal{D}$ be functors. For natural transformations $\phi : F \rightsquigarrow G$ and $\psi : G \rightsquigarrow H$, their vertical composition $\psi \circ_v \phi$ has as components the compositions of components of ϕ and ψ: $(\psi \circ_v \phi)_a = \psi_a \circ \phi_a$. The other, difficult one is horizontal composition.

Definition 4.2.4 (Horizontal composition of natural transformations)**.** Let $F, G : \mathcal{C} \to \mathcal{D}$ and $F', G' : \mathcal{D} \to \mathcal{E}$ be functors; $\phi : F \rightsquigarrow G$ and $\psi : F' \rightsquigarrow G'$ be natural transformations. For any object $a \in \mathcal{C}_0$, the morphisms $\psi_{G_0(a)} \circ F'_1(\phi_a)$ and $G'_1(\phi_a) \circ \psi_{F_0(a)}$ are equal, and define the horizontal composition $\psi \circ_h \phi$ by setting (either of) them to be the component for a:

$$(\psi \circ_h \phi)_a = \psi_{G_0(a)} \circ F'_1(\phi_a) \ \left(= G'_1(\phi_a) \circ \psi_{F_0(a)}\right).$$

□

At last, monads are defined using the notions presented above:

Definition 4.2.5 (Monad)**.** Let $\mathcal{C}$ be a category. A monad on $\mathcal{C}$ is a structure for the following signature:

- Base functor $M : \mathcal{C} \to \mathcal{C}$.
- Unit natural transformation $\eta : \mathrm{Id}_{\mathcal{C}} \rightsquigarrow M$.
- Multiplication natural transformation $\mu : M \circ M \rightsquigarrow M$.
- Left unit law: $\mu \circ_v (\eta \circ_h \mathrm{Id}_M) = \mathrm{Id}_M$, or written componentwise, $\mu_a \circ \eta_{M_0(a)} = \mathrm{id}_{M_0(a)}$.
- Right unit law: $\mu \circ_v (\mathrm{Id}_M \circ_h \eta) = \mathrm{Id}_M$, or $\mu_a \circ M_1(\eta_a) = \mathrm{id}_{M_0(a)}$.
- Associativity law: $\mu \circ_v (\mathrm{Id}_M \circ_h \mu) = \mu \circ_v (\mu \circ_h \mathrm{Id}_M)$, or $\mu_a \circ M_1(\mu_a) = \mu_a \circ \mu_{M_0(a)}$. □

4.3 Formalization of the theories of functors and monads

Our formalization of monads starts with the theory of functors. This is different from the hierarchy given in [Gibbons and Hinze, 2011], where they start directly from the definition of monads. However we find that separating out the theory of functors results in a simpler organization of lemmas used in equational reasoning and results in a more robust hierarchy.

As we noted in the introduction, we restrict ourselves to monads on **Set**. Thus the definition of functors is also restricted to **Set**.

Definition 4.3.1 (`Module Functor` in [Affeldt et al., 2020a, `monad.v`])**.** A functor on the category of sets $F : \mathbf{Set} \to \mathbf{Set}$ (or just F) is defined by the following signature:

- Action on objects $F_0 : V \to V$, where V is the class of all sets.
- Action on morphisms $F_1 : (a \to b) \to (F_0(a) \to F_0(b))$.
- Identity law: $F_1(\mathrm{id}_a) = \mathrm{id}_{F_0(a)}$.
- Composition law: $F_1(g \circ f) = F_1(g) \circ F_1(f)$. □

In the Coq code, this definition forms a packed class with the following mixin:

```
Record mixin_of (F0 : Type -> Type) : Type := Mixin {
  F1 : forall a b, (a -> b) -> F0 a -> F0 b ;
  _ : forall a, F1 id = id :> (F0 a -> F0 a) ;
  _ : forall a b c (g : b -> c) (h : a -> b),
        F1 (g \o h) = F1 g \o F1 h :> (F0 A -> F0 C) }.
```

As we noted in Section 2.5, the type universe `Type` in Coq represents a Grothendieck universe in the set-theoretic interpretation with Grothendieck universes. Therefore, the above mixin actually defines a functor on $\mathbf{Set}_{\mathcal{U}}$, the category of all elements in $\mathcal{U}$. This category is $\mathcal{U}^+$-small for some larger Grothendieck universe $\mathcal{U}^+$ and not equal to the large category **Set**. Within the scope of this thesis, however, the difference is not significant as we do not run into problems that are sensitive to such a size issue. Hence we are going to deliberately ignore the difference.

The formal definition of natural transformations is similar to Definition 4.2.3. Its core is the naturality law:

```
Definition P (phi : F ~~> G) :=
  forall a b (f : a -> b), (G # f) \o phi a = phi b \o (F # f).
```

where `F ~~> G` is a notation for the family of morphisms $(\phi_a : F_0(a) \to G_0(a))_a$, and `F # f`, `G # f` are $F_1(f)$, $G_1(f)$ respectively.

Monad on **Set** is also similar to Definition 4.2.5.

Definition 4.3.2 (`Module Monad` in [Affeldt et al., 2020a, `monad.v`])**.** A monad on **Set** is a structure for the following signature:

- Base functor $M : \mathbf{Set} \to \mathbf{Set}$.
- Unit natural transformation $\mathtt{ret} : \mathrm{Id}_{\mathbf{Set}} \rightsquigarrow M$.
- Multiplication natural transformation $\mathtt{join} : M \circ M \rightsquigarrow M$.
- Left unit law: $\mathtt{join}_a \circ \mathtt{ret}_{M_0(a)} = \mathrm{id}_{M_0(a)}$.

- Right unit law: $\texttt{join}_a \circ M_1(\texttt{ret}_a) = \mathrm{id}_{M_0(a)}$.
- Associativity law: $\texttt{join}_a \circ M_1(\texttt{join}_a) = \texttt{join}_a \circ \texttt{join}_{M_0(a)}$. □

In the formalization, the following mixin for monads is mixed with functors to define the packed class of monads:

```
Record mixin_of (M : functor) : Type := Mixin {
  ret : FId ~> M ;
  join : M \O M ~> M ;
  _ : forall a, @join a \o @ret (M a) = id ;
  _ : forall a, @join a \o M # @ret a = id ;
  _ : forall a, @join a \o M # @join a = @join a \o @join (M a) }.
```

The notation `F ~> G` denotes a family of morphisms (`F ~~> G`) equipped with the proof of naturality law for it.

Monads in programming languages are often utilized via `bind` operator and the "do" notations as the interface. We define `bind` in terms of `join`, and provide these notations which look almost identical to those found in Haskell programs.

- For $m \in M_0(a)$ and $f : a \to M_0(b)$, $\texttt{bind}(m, f) = \texttt{join}((M_1(f))(m))$.
- `(do x <- m ; e)` denotes $\texttt{bind}(m, (x \mapsto e))$.
- `(do x : T <- m ; e)` denotes $\texttt{bind}(m, ((x \in T) \mapsto e)$.
- `(m >>= f)` denotes $\texttt{bind}(m, f)$.
- `(m >> e)` denotes $\texttt{bind}(m, (x \mapsto e))$, where x does not freely occur in e.

In the Coq code, these definitions and the declarations of notations look like:

```
Definition Bind a b (m : M a) (f : a -> M b) : M b := Join ((M # f) m).
Notation "'do' x <- m ; e" := (Bind m (fun x => e)).
Notation "'do' x : T <- m ; e" := (Bind m (fun x : T => e)).
Notation "m >>= f" := (Bind m f).
Notation "m >> e" := (Bind m (fun _ => e)).
```

Remark. In the study of programming languages, it is usual that monads are defined directly in terms of `bind` instead of `join`. The reason we choose `join` is that it allows a modular separation between functors and monads. `bind` operator and its laws induce another functor structure, which will be redundant and conflicting given the inherited functor structure. Yet, it is often convenient to be able to define monads also by supplying `ret` and `bind` that satisfy appropriate laws. Such a facility is provided as `Module Monad_of_ret_bind` in [Affeldt et al., 2020a, `monad.v`]. □

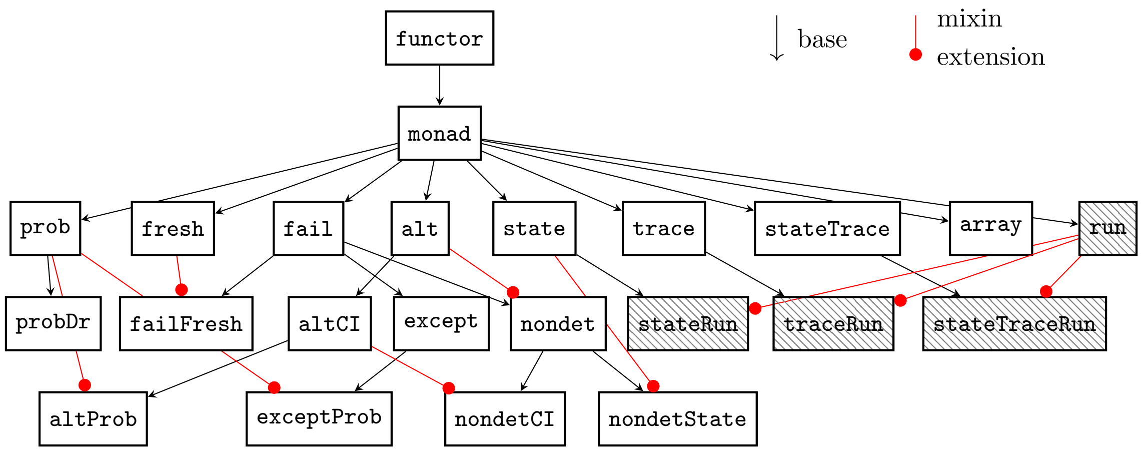


Figure 4.1: Hierarchy of effects formalized in [Affeldt et al., 2020a]

4.4 Extensions of the theory of monads

The heart of our formalization is a hierarchy of effects. As explained in Section 4.1, each effect is represented by a theory which extends the theory of monad with additional algebraic structure that defines the effect, providing effect operators and equational laws that capture the properties of effect.

Various theories of effects are formalized in our MONAE library. Figure 4.1 is a pictorial overview of those effects and their relations of extensions. [1] Among them, we are going to present three examples here: nondeterministic choice (`altCI`), probabilistic choice (`prob`), and combined choice (`altProb`).

4.4.1 Monads with nondeterministic choice

Our first example is the extension with nondeterministic choice. A program is said to be nondeterministic if it may answer many values for a single input. Such nondeterminisms may come from a parallel computation in a search program, or may be an undesirable race condition in many-agent systems. In programs, these different kinds of nondeterminism results are all expressed by binary operators (*choice operators*), but with different algebraic properties.

We show here two theories with different laws: `MonadAlt` for noncommutative nondeterministic choice with only associativity, and `MonadAltCI` with additionally commutativity and idempotence. The resulting theory `MonadAltCI` can be seen as a combination of the theories of semilattices and monads.

[1] Gray boxes indicate that the theories involve deep embedding.

Definition 4.4.1 (Semilattice)**.** A semilattice is a structure for the following signature:

- Carrier set X;
- Semilattice operation $\Box : X \times X \to X$;
- Idempotence law: $x \Box x = x$;
- Commutativity law: $x \Box y = y \Box x$;
- Associativity law: $(x \Box y) \Box z = x \Box (y \Box z)$. $\square$

Definition 4.4.2 (Module MonadAltCI in [Affeldt et al., 2020a, fail_monad.v])**.** A nondeterministic choice monad is a structure for the following signature:

- Base monad M;
- Semilattice operations $\Box_X : M_0(A) \times M_0(A) \to M_0(A)$ for any set A;
- Structure axiom: For any A, $M_0(A)$ is a semilattice with $\Box_A$ as its operation.
- Left-bind-distributivity law: $\texttt{bind}(x \Box_A y, f) = \texttt{bind}(x, f) \Box_A \texttt{bind}(y, f)$. $\square$

The noncommutative nondeterministic choice monad MonadAlt is its subtheory with the commutativity and idempotence laws omitted:

```
Record mixin_of (M : monad) : Type := Mixin {
  alt : forall A, M A -> M A -> M A ;
  _ : forall A, associative (@alt A) ;
  _ : BindLaws.left_distributive (@Bind M) alt
}.
```

Then nondeterministic choice monad MonadAltCI is recovered by adding commutativity and idempotence to recover the semilattice structure:

```
Record mixin_of (M : Type -> Type) (op : forall A, M A -> M A -> M A) : Type := Mixin {
  _ : forall A, idempotent (op A) ;
  _ : forall A, commutative (op A)
}.
```

4.4.2 Monads with probabilistic choice

Tne next example is probabilistic choice. Similarly to nondeterminism, a probabilistic choice in a programs is expressed by an operator choosing from two values, but in this case, according to a fixed probability. In other words, the probability that each value be taken is fixed at each probabilistic choice, while it is abstracted away and cannot be known at nondeterministic choices. This

probabilistic choice is exactly the convex combination in convex spaces 3.2, and the probabilistic choice monads are defined to be a convex space structure mixed with monad structure.

Definition 4.4.3 (`Module MonadProb` in [Affeldt et al., 2020a, `proba_monad.v`])**.** A probabilistic choice monad is a structure for the following signature:

- Base monad M;
- Convex combination operations $_\triangleleft p \triangleright_A _ : M_0(A) \times M_0(A) \to M_0(A)$ for any set A and $p \in [0, 1]$;
- Structure axiom: For any A, $M_0(A)$ is a convex space with $_\triangleleft_\triangleright_A_$ as its operation.
- Left-bind-distributivity law: $\mathtt{bind}(x \triangleleft p \triangleright_A y, f) = \mathtt{bind}(x, f) \triangleleft p \triangleright_A \mathtt{bind}(y, f)$. □

4.4.3 Monads with combined choice

We can combine the above two extensions into one, namely the *combined choice*. The combination of two choices means a style of programs that involves both of these operators, with a (semantic) distributivity law stating that probabilistic choice distributes over nondeterministic choice. The resulting combined choice is used in representing probabilistic systems that depend on nondeterministic inputs.

The combined choice contains both of two choice operators from nondeterministic and probabilistic choices. These two operators interact via the distributivity of probability over nondeterminism:

$$x \triangleleft p \triangleright (y \square z) = (x \triangleleft p \triangleright y) \square (x \triangleleft p \triangleright z).$$

Remark. One could instead introduce the other distributivity law (which is also called the *dual distributivity law* [Mislove et al., 2004]):

$$x \square (y \triangleleft p \triangleright z) = (x \square y) \triangleleft p \triangleright (x \square z).$$

This law is given an interpretation in the study of process algebras [Yi and Larsen, 1992, Morgan et al., 1996].

We however do not deal with it as it has undesirable consequences [Gibbons, 2012, Section 5.2]. For example, if one employs both distributivity laws, the probabilistic choice collapses: $x \triangleleft p \triangleright y = x \triangleleft q \triangleright y$ for any $p, q \in]0, 1[$. This is clearly what we do not want as a model of probabilistic choice. □

Definition 4.4.4 (`Module MonadAltProb` in [Affeldt et al., 2020a, `proba_monad.v`])**.** A combined choice monad is a structure for the following signature:

- Base monad M;

- Semilattice operations $\Box_X : M_0(A) \times M_0(A) \to M_0(A)$ for any set A;
- Semilattice structure axiom: For any A, $M_0(A)$ is a semilattice with $\Box_A$ as its operation.
- Convex combination operations $_\triangleleft p \triangleright_A _ : M_0(A) \times M_0(A) \to M_0(A)$ for any set A and $p \in [0,1]$;
- Convex space structure axiom: For any A, $M_0(A)$ is a convex space with $_\triangleleft_\triangleright_A_$ as its operation.
- Left-distributivity law: $x \triangleleft p \triangleright_y (\Box z) A = (x \triangleleft p \triangleright_A y) \Box_A (x \triangleleft p \triangleright_A z)$.
- Left-bind-distributivity law: $\texttt{bind}(x \Box_A y, f) = \texttt{bind}(x, f) \Box_A \texttt{bind}(y, f)$. □

4.5 Examples of equational reasoning

In this section, we show some examples of equational reasoning on monadic effects. Further examples will be found in files of [Affeldt et al., 2020a], as listed in [Affeldt et al., 2019, Section 4.4].

4.5.1 rev_insert (taken from [Mu, 2019])

An example of nondeterministic choice is the insertion of an element to a sequence. Given a sequence s of length n and an element a, there are $n+1$ possible positions for a to be inserted in s. We define an insertion function that returns the possible cases nondeterministically.

We fix a set A and a nondeterministic choice monad M in this subsection.

Definition 4.5.1 (`insert` in [Affeldt et al., 2020a, `fail_monad.v`]).

$$\begin{array}{ccl} A \times A^{<\omega} & \to & M_0(A^{<\omega}) \\ (a, \epsilon) & \mapsto & \texttt{ret}_M(a\epsilon) \\ (a, s_0 \dots s_n) & \mapsto & \texttt{ret}_M(as_0 \dots s_n) \Box M_1(t \mapsto s_0 t)(\text{insert}(a, s_1 \dots s_n)) \end{array}$$

$A^{<\omega}$ denotes the set of words (finite sequences of elements of A) $\bigcup_{i \in \mathbb{N}} A^i$. ϵ is the empty word. Other words are expressed simply by juxtaposing elements and words like $a\epsilon$, $s_0 \dots s_n$, and $as_0 \dots s_n$.

In the second line of case analysis, two values are connected by the nondeterministic choice operator $\Box$. The left hand side $\texttt{ret}_M(as_0 \dots s_n)$ is a prefixing of a to $s_0 \dots s_n$. In the right hand side, the head s_0 of the word is removed and insert is applied to the remaining part $s_1 \dots s_n$. Each of the nondeterministic results is then prefixed by s_0 again.

The corresponding Coq definition is:

fmapE	:	$M_1(f)(m) = \mathtt{bind}_M(m, \mathtt{ret}_M \circ f)$
bindretf	:	$\mathtt{bind}_M(\mathtt{ret}_M(a), f) = f(a)$
bindmret	:	$\mathtt{bind}_M(m, \mathtt{ret}_M) = m$
prob_bindDl	:	$\mathtt{bind}_M(x \triangleleft p \triangleright y, f) = \mathtt{bind}_M(x, f) \triangleleft p \triangleright \mathtt{bind}_M(y, f)$
choicemm	:	$a \triangleleft p \triangleright a = a$
choiceC	:	$a \triangleleft p \triangleright b = b \triangleleft 1 - p \triangleright a$
convACA	:	$(a \triangleleft q \triangleright b) \triangleleft p \triangleright (c \triangleleft q \triangleright d) = (a \triangleleft p \triangleright c) \triangleleft q \triangleright (b \triangleleft p \triangleright d)$
alt_bindDl	:	$\mathtt{bind}_M(x \square_A y, f) = \mathtt{bind}_M(x, f) \square_A \mathtt{bind}_M(y, f)$
altC	:	$a \square_A b = b \square_A a$

Table 4.1: Lemmas used in formal proofs

```
Fixpoint insert (a : A) (s : seq A) : M (seq A) :=
  match s with
  | nil    => Ret (a :: nil)
  | h :: t => Ret (a :: h :: t) [~] fmap (fun s => h :: s) (insert a t)
  end.
```

The property we want to prove in this subsection is that insert commutes with rev, the reversing function:

$$(M_1(rev))(\text{insert}(a, s)) = \text{insert}(a, \text{rev}(s)).$$

The proof proceeds by induction on s.

```
Lemma rev_insert : rev (o) insert a = insert a \o rev :> (_ -> M _).
Proof.
rewrite boolp.funeqE; elim => [|h t IH].
  by rewrite fcompE insertE fmapE bindretf.
rewrite fcompE insertE compE alt_fmapDl fmapE bindretf compE [in RHS]rev_cons insert_rcons.
rewrite rev_cons -cats1 rev_cons -cats1 -catA; congr (_ [~] _).
move: IH; rewrite fcompE [X in X -> _]/= => <-.
rewrite -!fmap_oE; congr (fmap _ (insert a t)).
by rewrite boolp.funeqE => s; rewrite /= -rev_cons.
Qed.
```

We also show how the equational transformation goes on for the base case $s = \epsilon$ (appearing as

the line `by rewrite fcompE insertE fmapE bindretf.` above).

$$
\begin{array}{llll}
 & \text{LHS} & & \text{RHS} \\
= & (M_1(\text{rev}))(\text{insert}(a,\epsilon)) & = & \text{insert}(a,\text{rev}(\epsilon)) \\
= & (M_1(\text{rev}))(\texttt{ret}(a\epsilon)) & = & \text{insert}(a,\epsilon) \\
= & \texttt{bind}(\texttt{ret}(a\epsilon),\texttt{ret}\circ\text{rev}) \quad (\texttt{fmapE}) & = & \texttt{ret}(a\epsilon) \\
= & \texttt{ret}(\text{rev}(a\epsilon)) \quad (\texttt{bindretf}) & & \\
= & \texttt{ret}(a\epsilon) & &
\end{array}
$$

The induction step case is more involved and uses the lemma `insert_rcons` whose proof relies on the commutativity of nondeterministic choice.

4.5.2 two_coins

Think of tossing a biased coin which has probability p for it to land with the face side up. Such coins can be modeled using a probabilistic choice monad M as follows:

$$\texttt{bcoin(p)} = \texttt{ret}(\text{true}) \triangleleft p \triangleright \texttt{ret}(\text{false}).$$

```
Definition bcoin {M : probMonad} (p : prob) : M bool :=
  Ret true <| p |> Ret false.
```

Now there are two such coins with biases p and q and you toss them one by one, recording the ordered pair of the results.

```
Definition two_coins : M (bool * bool)%type :=
  (do a <- bcoin p; (do b <- bcoin q; Ret (a, b) : M _))%Do.

Definition two_coins' : M (bool * bool)%type :=
  (do a <- bcoin q; (do b <- bcoin p; Ret (b, a) : M _))%Do.
```

In `two_coins`, you toss first p-coin with the result a then q-coin with b, and record (a, b). In `two_coins'`, the order of tossing coins and the order in the recorded pair are inverted. These recorded values form probabilistic distributions over the set $\{\text{true}, \text{false}\} \times \{\text{true}, \text{false}\}$. We can show by equational reasoning on M that `two_coins` and `two_coins'` are the same distributions.

```
Lemma two_coinsE : two_coins = two_coins'.
Proof.
rewrite /two_coins /two_coins' /bcoin.
rewrite prob_bindDl.
rewrite !bindretf.
```

```
rewrite !(prob_bindDl,bindretf).
apply (@convex_choice.convACA probConvex).
Qed.
```

The equational transformation along the proof script looks like this:

$$
\begin{array}{lll}
 & \text{LHS} & \\
= & \texttt{two_coins} & \\
= & \texttt{(do a <- bcoin p; (do b <- bcoin q; Ret (a, b) : M _))} & \\
= & \texttt{(do a <- Ret true <|p|> Ret false; (do b <- bcoin q; Ret (a, b) : M _))} & \\
= & \texttt{(do a <- Ret true; (do b <- bcoin q; Ret (a, b) : M _))} \triangleleft p \triangleright (\ldots) & (\texttt{prob_bindDl}) \\
= & (\texttt{do b <- bcoin q; Ret (true, b) : M _}) \triangleleft p \triangleright (\ldots) & (\texttt{bindretf}) \\
= & (\texttt{ret}(\text{true}, \text{true}) \triangleleft q \triangleright \texttt{ret}(\text{true}, \text{false})) \triangleleft p \triangleright & \\
 & (\texttt{ret}(\text{false}, \text{true}) \triangleleft q \triangleright \texttt{ret}(\text{false}, \text{false})) & (\texttt{!(prob_bindDl,bindretf)}) \\
= & (\texttt{ret}(\text{true}, \text{true}) \triangleleft p \triangleright \texttt{ret}(\text{false}, \text{true})) \triangleleft q \triangleright & \\
 & (\texttt{ret}(\text{true}, \text{false}) \triangleleft p \triangleright \texttt{ret}(\text{false}, \text{false})) & (\texttt{convACA})
\end{array}
$$

The right hand side arrives at the same expression by exactly the same transformations except for the last `convACA`, hence the proof is done.

4.5.3 arbcoin and coinarb (taken from [Gibbons and Hinze, 2011])

In this section, we show a tiny example that utilizes the combined choice monad. The `arb` denotes a simple arbitrary choice, which evaluates to either true or false nondeterministically.

$$\texttt{arb} = \texttt{ret}(\text{true}) \square \texttt{ret}(\text{false})$$

It is a bit interesting when we sequence `bcoin` and `arb`, the `bcoin` part disappears. This result crucially depends on the commutativity of nondeterministic choice, which is inherited to the combined choice monad from the (commutative) nondeterministic monad. Here are the proof script and the corresponding equational transformations.

```
Definition coinarb p : M bool :=
  (do c <- bcoin p ; (do a <- arb; Ret (a == c) : M _))%Do.

Lemma coinarb_spec p : coinarb p = arb.
Proof.
rewrite [in LHS]/coinarb [in LHS]/bcoin.
rewrite prob_bindDl.
rewrite 2!bindretf.
```

```
rewrite /arb !alt_bindDl !bindretf !eqxx.
by rewrite eq_sym altC choicemm.
Qed.
```

$$
\begin{aligned}
& \texttt{coinarb(p)} & \\
= \;& \texttt{bind}(\texttt{ret}(\text{true}), c \mapsto \texttt{bind}(\text{arb}, a \mapsto \texttt{ret}(a = c))) \triangleleft p \triangleright & (\texttt{prob_bindDl}) \\
& \texttt{bind}(\texttt{ret}(\text{false}), c \mapsto \texttt{bind}(\text{arb}, a \mapsto \texttt{ret}(a = c))) & \\
= \;& \texttt{bind}(\text{arb}, a \mapsto \texttt{ret}(a = \text{true})) \triangleleft p \triangleright \texttt{bind}(\text{arb}, a \mapsto \texttt{ret}(a = \text{false})) & (\texttt{!bindretf}) \\
= \;& \texttt{ret}(\text{true} = \text{true}) \,\Box\, \texttt{ret}(\text{false} = \text{true}) \triangleleft p \triangleright & (\texttt{!alt_bindDl}, \texttt{!bindretf}) \\
& \texttt{ret}(\text{true} = \text{false}) \,\Box\, \texttt{ret}(\text{false} = \text{false}) & \\
= \;& \texttt{ret}(\text{true}) \,\Box\, \texttt{ret}(\text{false}) \triangleleft p \triangleright \texttt{ret}(\text{false}) \,\Box\, \texttt{ret}(\text{true}) & \\
= \;& \texttt{ret}(\text{true}) \,\Box\, \texttt{ret}(\text{false}) \triangleleft p \triangleright \texttt{ret}(\text{true}) \,\Box\, \texttt{ret}(\text{false}) & (\texttt{altC}) \\
= \;& \texttt{ret}(\text{true}) \,\Box\, \texttt{ret}(\text{false}) & (\texttt{choicemm}) \\
= \;& \texttt{arb} &
\end{aligned}
$$

On the other hand, if we sequence them in a different order, first `arb` then `bcoin`, the result is different.

```
Definition arbcoin p : M bool :=
  (do a <- arb ; (do c <- bcoin p; Ret (a == c) : M _))%Do.

Lemma arbcoin_spec p :
  arbcoin p = (bcoin p : M _) [~] bcoin p.~%:pr.
Proof.
rewrite /arbcoin /arb.
rewrite alt_bindDl 2!bindretf.
rewrite bindmret; congr (_ [~] _).
rewrite [in RHS]/bcoin choiceC.
rewrite [in RHS](@choice_ext p); last by rewrite /= onemK.
by rewrite {1}/bcoin prob_bindDl 2!bindretf eqxx /=.
Qed.
```

$$
\begin{array}{lll}
& \texttt{arbcoin(p)} & \\
= & \texttt{bind}(\texttt{bcoin(p)}, c \mapsto \texttt{ret}(\text{true} = c)) \,\square & (\texttt{prob_bindDl}, !\texttt{bindretf}) \\
& \texttt{bind}(\texttt{bcoin(p)}, c \mapsto \texttt{ret}(\text{false} = c)) & \\
= & \texttt{bcoin(p)} \,\square\, \texttt{bind}(\texttt{bcoin(p)}, c \mapsto \texttt{ret}(\text{false} = c) & (\texttt{bindmret}) \\
= & \texttt{bcoin(p)} \,\square\, \texttt{ret}(\text{false}) \triangleleft p \triangleright \texttt{ret}(\text{true}) & (\texttt{prob_bindDl}, !\texttt{bindretf}) \\
= & \texttt{bcoin(p)} \,\square\, \texttt{ret}(\text{false}) \triangleleft 1 - (1-p) \triangleright \texttt{ret}(\text{true}) & \\
= & \texttt{bcoin(p)} \,\square\, \texttt{ret}(\text{true}) \triangleleft (1-p) \triangleright \texttt{ret}(\text{true}) & (\texttt{choiceC}) \\
= & \texttt{bcoin}(p) \,\square\, \texttt{bcoin}(1-p) &
\end{array}
$$

4.6 Models

We have so far presented the formalization of the extended theories of monads and the method of equational reasoning based on those theories. This theory-driven analysis on effects is straightforward from the viewpoint of programming languages, since theories are rather syntactical, finitary objects. For this reason, theories retain good affinity to the formalization. As tools of proofs, theories provide clean proofs of the positive results: which equations are satisfied for the effect of interest.

On the other hand, models are often harder to formalize than theories since they usually involve infinitary mathematics such as analysis or geometry. Models are however indispensable to seriously understand theories with intuitions and assurance of the consistency. Models are also good at revealing negative results such as which equations are never derived from the axioms.

In this section, we show two examples of the construction of models for the theories of effects. They are again for the nondeterministic and probabilistic choice monads.

4.6.1 Semilattice model for `MonadAltCI` [Affeldt et al., 2020a, `monad_model.v`]

We look first for a model M of the theory of nondeterministic choice monad. M should endow every set X with a semilattice structure $M_0(X)$ and that is actually all that is needed. One easy example of it is the powerset monad (`Module ModelAltCI` in [Affeldt et al., 2020a, `monad_model.v`]).

4.6.2 Distribution model for `MonadProb` [Affeldt et al., 2020a, `proba_monad_model.v`]

We want next a model M of the theory of probabilistic choice monad. As in the case of nondeterministic choice, what M has to do is to provide a convex space structure $M_0(X)$ for every set X. This is achieved by taking all finitely-supported distributions over X (`Module MonadProbModel` in [Affeldt et al., 2020a, `proba_monad_model.v`]).

4.6.3 Nonempty convex powerset model for `MonadAltProb`

Although the previous two examples have been fairly easy, the model construction of the theory of combined choice monad turns out a very different, difficult problem. This is because the definitions needed to construct the monad does not always take place in the category of sets. The structure we employ is the nonempty convex powersets in Section 3.7. This construction is based on another category, the category of convex spaces, which demands a further generalized framework of category theory for handling. That is the topic of the next chapter, where an example of the combined choice monad is achieved.

Chapter 5

Equational Reasoning on Categories

5.1 Introduction

In this chapter, we answer the problem posed in Section 4.6.3, that is to construct a model of the theory of monads with combined choice (`MonadAltProb`). As explained in Section 3.7, combined choice describes the nondeterminism of programs that involve both nondeterministic and probabilistic choices with the latter distributing over the former.

The theories of monads with either of these choices (Sections 4.4.2 and 4.4.1) have already been provided with models (monads) in Sections 4.6.2 and 4.6.1. As there are several general methods to combine monads, one might expect that the monad for combined choice may be obtained straightforwardly from them. Among such methods, the combination based on monadic distributivity law is a candidate to obtain the desired monad, unfortunately with a failure in the distributivity [Varacca and Winskel, 2006, Proposition 3.2].

Instead, we can turn our attention to the nonempty convex powerset construction $\mathscr{P}_c$, which was introduced as a structure for combined choice. The goal would be achieved if we could turn it into a monad in our semantics, but $\mathscr{P}_c$ takes a convex space as argument, not an arbitrary set. If we simply endow $\mathscr{P}_c$ with a monad structure, the resulting monad is that on the category **Conv** of convex spaces. This is different from what we want, a monad on the category **Set** of sets.

This problem can be amended by first lifting a given set X to the set of distributions over X, which form a convex space. It is essentially the same as what the probabilistic monad does and we can reuse most ideas from it for this part.

The action of the desired monad has now two steps: taking the set of distributions and taking its nonempty convex powerset. These steps are known to form adjunctions: [Cheung, 2017, Propositions 6.3.3 and 6.3.4] [Varacca and Winskel, 2006, Proposition 5.7]

Figure 5.1 is the overview of these adjunctions. **C** is the "category of combined choice" which

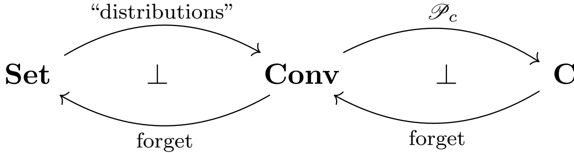


Figure 5.1: Overview of the adjunctions.

we will identify later. Nonempty convex powersets should live in **C** as objects.

Since adjoint functors compose again into an adjoint functor, and any adjunction gives rise to a monad, it seems to be a good idea to formalize these adjunctions. This is actually the route we go through in this chapter.

So far, we have dealt only with those monads on **Set** and did not formalize other categories such as **Conv** nor the framework to deal with other categories **Set**. In order to state adjunctions, we must first work on some category theory.

The design principle of our formalization of category theory is twofold:

1. It should be compatible with shallow embedding and equational reasoning.
2. Especially, it should be useful as an auxiliary device for defining monads on **Set**.

Principle 1 leads to a formalization of *concrete categories* rather than abstract ones, that is, objects are certain sets and morphisms are functions among them. For principle 2, we provide a conversion layer to regard the monads on **Set** of this chapter as those of Chapter 4.

5.2 Formalization of Categories

As explained in the introduction, our formalization of categories is specialized to concrete ones. For the smooth introduction to this topic, we start with recollecting the definition of abstract category given in Section 4.2:

Definition (Abstract category, (Definition 4.2.1))**.** An abstract category $\mathcal{C}$ is a structure for the following signature:

- Class of objects $\mathcal{C}_0$.
- Classes of morphisms $\hom_{\mathcal{C}}(a, b)$ for $a, b \in \mathcal{C}_0$.
- Identity morphisms $\mathrm{id}_a : a \to a$ for every $a \in \mathcal{C}_0$.

- Compositions of morphisms $g \circ f : a \to c$ for every $a, b, c \in \mathcal{C}_0$, $f : a \to b$, and $g : b \to c$.
- Unit law: $\mathrm{id}_b \circ f = f \circ \mathrm{id}_a = f$ for any $f : a \to b$.
- Associativity law: $h \circ (g \circ f) = (h \circ g) \circ f$ for any $a, b, c, d \in \mathcal{C}_0$, $f : a \to b$, $g : b \to c$, and $h : c \to d$. □

In our definition of concrete categories, objects are sets indexed by some set T and morphisms are functions between those indexed sets:

Definition 5.2.1 (Module Category in [Affeldt et al., 2020a, category.v])**.** A concrete category is a structure for the following signature:

- Indexing set T.
- Object specifier function $\mathcal{C}_0 : T \to \mathcal{U}$, where $\mathcal{U}$ is a Grothendieck universe of sets.
- Hom-sets $\mathrm{hom}(A, B) \subseteq (\mathcal{C}_0(A) \to \mathcal{C}_0(B))$ for $A, B \in T$.
- Identity law: $\mathrm{id}_A \in \mathrm{hom}(A, A)$ for any $A \in T$, where id_A is the identity function on $\mathcal{C}_0(A)$.
- Composability law: if $f : \mathcal{C}_0(A) \to \mathcal{C}_0(B)$ is in $hom(A, B)$ and $g : \mathcal{C}_0(B) \to \mathcal{C}_0(C)$ in $hom(B, C)$, then $(g \circ f) \in hom(A, C)$, for any $A, B, C \in T$. □

One may notice that the unit and associativity laws present in the definition of abstract categories are not stated for the concrete ones. Since the identity morphisms are indeed identity functions on sets and compositions are those of functions, the necessary laws are automatically satisfied. In other words, they are inherited from **Set**.

This definition is then formalized as a packed class with the following mixin:

```
Record mixin_of (T : Type) : Type := Mixin {
  obj : T -> Type ;
  hom : forall A B, (obj A -> obj B) -> Prop ;
  _ : forall (A : T), hom (A:=A) (B:=A) id ;
  _ : forall (A B C : T) (f : obj A -> obj B) (g : obj B -> obj C),
      hom f -> hom g -> hom (g \o f) ;
}.
```

Remark. In the literature, a concrete category is often defined to be a category with a faithful[1] functor to **Set** [Freyd, 1973]. Our definition is equivalent to this one: the object specifier becomes the object part of the functor, and the faithfulness is expressed by the inclusion of hom-sets. □

[1] A functor is said to be faithful if they are injective on each hom-set.

5.3 Examples of formalized categories

We list examples of the categories formalized in our library. Actually, these categories are not just examples but are all used to later construct the model of the theory of combined choice.

5.3.1 Category of sets

`Section Type_as_a_category` in [Affeldt et al., 2020a, `category.v`]

The first, easiest-to-define example is the category **Set** of sets. As we are working in the set-theoretic interpretation, this category implicitly exists without any extra definition. We just prepare a thin layer to make it look like our concrete category. The object specifier is an identity function and the hom-sets are just always-true predicates.

```
Definition Type_category_class : Category.mixin_of Type :=
  @Category.Mixin Type id (fun _ _ _ => True) (fun _ => I) (fun _ _ _ _ _ _ _ => I).
```

5.3.2 Category of sets with choice functions

`Section choiceType_as_a_category` in [Affeldt et al., 2020a, `gcm_model.v`]

Next we pack the sets with choice functions as a category. A choice function on a set X enables one to algorithmically choose elements from any set of nonempty subsets of X. The axiom of choice we employ assures that every set has its canonical choice function. With this form of the axiom of choice and the coercion from sets with choice functions to usual sets, we obtain the category **ChSet** of sets with choice functions almost as easily as the category of sets.

```
Definition choiceType_category_mixin : Category.mixin_of choiceType :=
  @Category.Mixin choiceType id (fun _ _ _ => True) (fun _ => I) (fun _ _ _ _ _ _ _ => I).
```

5.4 Affine functions and the category of convex spaces

`Section convType_as_a_category` in [Affeldt et al., 2020a, `gcm_model.v`]

We define a category structure on convex spaces from Section 3.2. Like any algebra, convex spaces come with an appropriate notion of homomorphisms, namely *affine functions*, that preserve convex combination operations.

Definition 5.4.1 (`affine_function_at` in [Infotheo, 2020, `probability/convex_choice.v`])**.** A function $f : A \to B$ between convex spaces A and B is affine if, for any $p \in [0, 1]$ and $x, y \in A$, $f(x \triangleleft p \triangleright y) = f(x) \triangleleft p \triangleright f(y)$ holds. □

The identity maps on convex spaces are affine and compositions of affine functions are again affine. Therefore, the set of convex spaces and the set of affine functions form a category, which we call the category of convex spaces and denote by **Conv**.

```
Definition convType_category_mixin : Category.mixin_of convType :=
  Category.Mixin affine_function_id_proof affine_function_comp_proof'.
```

5.5 Formalization of monads defined from adjunctions

The combined choice monad as in Cheung [Cheung, 2017] is defined in terms of adjunctions. We are going to follow this method and therefore need to formalize adjunctions.

Definition 5.5.1 (`Module AdjointFunctor` in [Affeldt et al., 2020a, `category.v`])**.** Let $F : \mathcal{C} \to \mathcal{D}$ and $G : \mathcal{D} \to \mathcal{C}$ be functors. An adjunction of F and G is a structure for the following signature.

- Unit natural transformation $\eta : \mathrm{Id} \rightsquigarrow G \circ F$.
- Counit natural transformation $\epsilon : F \circ G \rightsquigarrow \mathrm{Id}$.
- Left triangular law: $\epsilon_{F_0(c)} \circ F_1(\eta_c) = \mathrm{id}$.
- Right triangular law: $G_1(\epsilon_d) \circ \eta_{G_0(d)} = \mathrm{id}$. □

We say F is *left-adjoint* to G and G *right-adjoint* to F.

For such an adjunction, a monad M is defined by setting $M = G \circ F$. The unit of M is the same as that of adjunction. The multiplication of M is obtained by using the counit ϵ:

$$\mu = (G \circ_h \epsilon \circ_h F) : G \circ F \circ G \circ F \rightsquigarrow G \circ F.$$

The details of this construction is given in `Module MonadOfAdjoint` of [Affeldt et al., 2020a, `category.v`].

We also have to compose adjunctions into an adjunction when we define a monad from adjunctions in Figure 5.1. It is proved in `Module AdjComp` of [Affeldt et al., 2020a, `category.v`].

5.6 Semicomplete semilattice

We are planning to turn the nonempty convex set construction $\mathscr{P}_c$ into the combined choice monad, via adjunctions in Figure 5.1. In the diagram, we haven't yet determined the category **C**. It should be a category where nonempty convex sets live in. In this section, we define the theory of semicomplete semilattices. It is going to be combined with convex spaces in the next section to obtain the algebraic theory for nonempty convex sets.

In our model of combined choice, the nondeterministic choice operation is realized by an infinitary convex powerset operation, rather than the finitely-generated one as in Cheung's PhD thesis [Cheung, 2017]. An algebraic theory that contains this operation is the theory of semicomplete semilattices, which is an extension of the theory of semilattices (Section 4.4.1).

A semicomplete semilattice is a structure for the following signature [Beaulieu, 2008, Definition 3.2.3]:

- Carrier set X;
- Infinitary semilattice operation $\square : \mathscr{P}_+(X) \to X$, where $\mathscr{P}_+(X) = \{S \subseteq X \mid S \neq 0\}$;
- Singleton law: For any element x of X, $\square\{x\} = x$;
- Replacement-union law: For any set I, nonempty subset S of I, and function $F : I \to \mathscr{P}_+(X)$,

$$\square \left(\bigcup_{x \in S} F(x) \right) = \square_{x \in S} \left(\square F(x) \right),$$

where $\square_{x \in S} (\ldots x \ldots)$ is the abbreviation for $\square \{\ldots x \ldots \mid x \in S\}$.

The replacement-union law can equivalently be decomposed into two simpler laws that appear in [Reiterman, 1986]:

- $\square_{i \in k} x_{f(i)} = \square_{j \in f[k]} x_j$;
- $\square_{i \in n} \left(\square_{j \in k_i} x_j \right) = \square_{j \in \left(\bigcup_{i \in n} k_i \right)} x_j$.

5.7 Semicomplete semilattice convex space

5.7.1 Definition

In the previous section, we have defined the theory of semicomplete-semilattice, which models the nondeterminism, constituting a half of our goal. In order to obtain a combined model of nondeterministic and probabilistic choices, we need to mix the theories of semicomplete semilattices and convex spaces into a combined algebra, which we call *semicomplete semilattice convex space*.

When mixing, we require additional law to hold in the combined theory. This is an infinitary version of the one present in Gibbons and Hinze [Gibbons and Hinze, 2011, Section 8.2] (see Section 4.4.3 for additional explanations), namely a distributivity between operations [Infotheo, 2020, `Lemma JoetDr`, file `necset.v`]: Let X be the carrier set. For any $p \in [0,1]$, $x \in X$, and nonempty $Y \subseteq X$,

$$x \triangleleft p \triangleright \left(\square Y \right) = \square \{ x \triangleleft p \triangleright y \mid y \in Y \}.$$

5.7.2 Technical lemmas

An important technical lemma about semicomplete semilattice convex spaces is that the operation is compatible with the operation of taking convex hulls.

Lemma 5.7.1. *For any nonempty set X in a semicomplete semilattice convex space,*

$$\bigsqcup \overline{X} = \bigsqcup X. \quad \Box$$

First, we lift the operator of convex spaces ($\triangleleft p \triangleright$) from points to sets of points; we denote this lifted operator by ($: \triangleleft p \triangleright :$). We use this lifted operator to define a new binary operator $X : \Box : Y := \bigcup_{p \in [0,1]} X : \triangleleft p \triangleright : Y$. Second, we show that

$$\overline{X} = \bigcup_{i \in \mathbb{N}} \underbrace{X : \Box : X : \Box : \ldots : \Box : X}_{i+1 \text{ occurrences of } X}.$$

Then, we show that

$$\Box(X) = \Box(X : \Box : X : \Box : \ldots : \Box : X),$$

using the property introduced by semicomplete semilattice convex spaces.

This lemma plays a critical role in proving the properties needed to define the monad for combined choice. It also has its corollaries the technical results from Varacca and Winskel's work (e.g., [Varacca and Winskel, 2006, Lemma 5.6]) and from Beaulieu's work (e.g., [Beaulieu, 2008, p. 56, l. 3]).

5.7.3 Category of semicomplete semilattice convex space

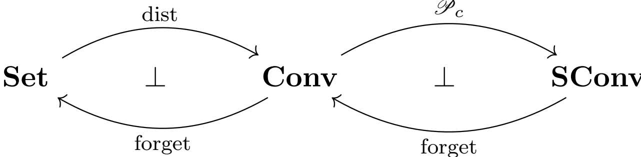
For semicomplete semilattice convex spaces, the appropriate notion of morphism is the one which is at the same time affine functions and preserving the semicomplete semilattice operations. We call such functions `JoetAffine` (`Module JoetAffine` [Infotheo, 2020, `necset.v`]).

A semicomplete semilattice convex spaces and `JoetAffine` functions form a category which we call **SConv** (`Section semiCompSemiLattConvType_as_a_category` in [Affeldt et al., 2020a, `category.v`]).

By ignoring the added semicomplete-semilattice structure, there is a forgetful functor from **SConv** to **Conv**. We show that $\mathscr{P}_c$ (Section 3.7) is left-adjoint to the forgetful functor (`Section eps1_eta1` in [Affeldt et al., 2020a, `gcm_model.v`]).

5.8 Construction of a model for the combined choice

So far we have obtained two levels of adjunctions, defining **SConv** and completing Figure 5.1:

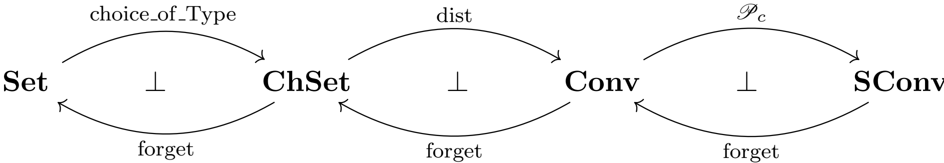


By applying the results in Section 5.5, we can obtain a monad from this diagram:

$$\text{forget} \circ \text{forget} \circ \mathscr{P}_c \circ \text{dist}.$$

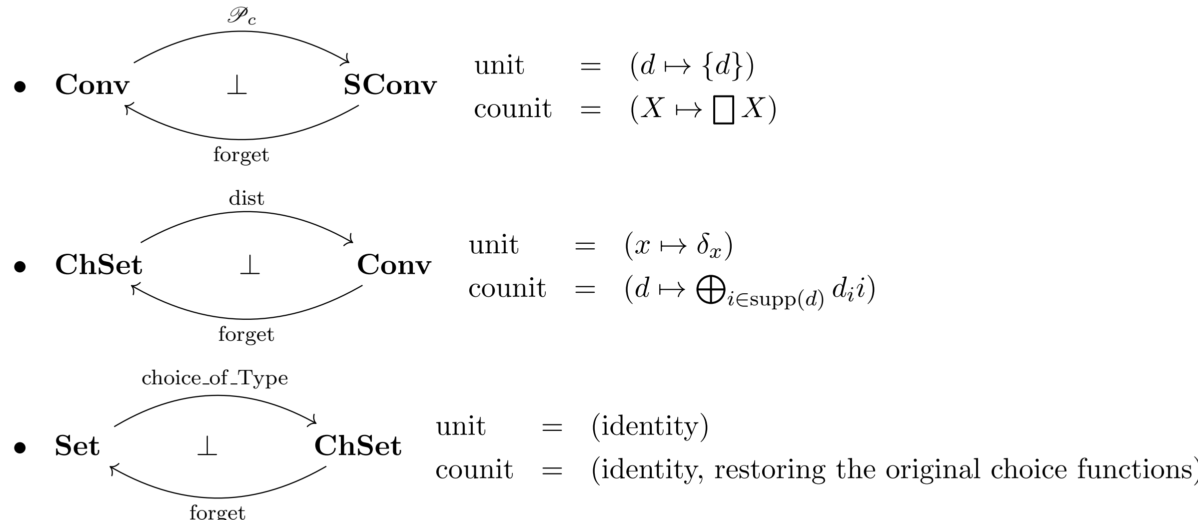
However, here remains a nuisance. The dist functor above is actualy a finitely-supported distribution functor defined over sets with choice functions. So the monad we have just obtained is not a monad on **Set** but on **ChSet**.

In order to fix this issue, we prefix another adjunction to the diagram:



The functor choice_of_Type attaches to every set a canonical choice function, thus it is induced from the axiom of choice.

The units and counits of these adjunctions are summarized as follows:



Our desired monad is finally constructed (`m` in [Affeldt et al., 2020a, `gcm_model.v`]):

$$\text{forget} \circ \text{forget} \circ \text{forget} \circ \mathscr{P}_c \circ \text{dist} \circ \text{choice_of_Type}.$$

5.9 Related work

5.9.1 Some history

The problem of this chapter has been a formalized construction of a combined model of probabilistic and nondeterministic choice monads. This problem has a fairly long history. An early account was in the study of *Communicating Sequential Processes* (CSP) specifically for the semantics of the probabilistic extension of CSP.

Though the semantics of probabilistic extensions of programming languages had been already modeled by Jones and Plotkin using the powerdomain construction [Jones and Plotkin, 1989], it did not admit an immediate composition with nondeterminism. Therefore the early attempts to apply the powerdomain construction to CSP did not go well [Mislove, 2000].

Mislove, and several others succeeded in constructing models in more abstract, purified settings. Varacca and Winskel [Varacca and Winskel, 2006], and Cheung [Cheung, 2017] rephrased the combined choice as the combination problem of monads.

5.9.2 Comparison to Cheung's finitely generated model

We have formalized the combined choice monad using semicomplete semilattice structure (with the infinitary operator). Finitary semilattice structure also works for the same purpose [Cheung, 2017, Varacca and Winskel, 2006]. Van Heerdt et al. [van Heerdt et al., 2018, p.4, footnote 5] and Bonchi et al. [Bonchi et al., 2017, Section 5.1] mention that either finitary or infinitary semilattice structures suffices for building a nonempty convex set monad.

Cheung proved his model to be a free model with respect to the combined choice, thus every property that holds in the model is inherited to any other model. Compared to it, our model is more specific and does not serve that purpose. We are expecting and seeking an appropriate notion of stronger nondeterminism to which our model is free and has the same role.

Cheung's main mathematical tools are sum and tensor combinations of monads which are mentioned in his thesis [Cheung, 2017, Chapter 3]. This is also a topic of Goncharov and Schröder's paper [Goncharov and Schröder, 2011]. The latter mentions the combinability according to the "ranks" of effects.

5.9.3 Comparison to domain-theoretic combination of nondeterminisms

Originally the semantical study of nondeterminism was sought in the context of domain-theoretic settings, or in the category of ω-complete partial orders. Three variants of powerdomain constructions were introduced: Smyth powerdomain, Hoare powerdomain, and Plotkin (convex) powerdomain [Plotkin, 1976, Plotkin, 1983]. Plotkin powerdomain subsumes the other two as the extreme

cases. These variants of powerdomains provide a neat interpretation of modal operators, as mentioned in [Winskel, 1983].

Thereafter, combining probabilistic and nondeterministic choices in powerdomain has been a research topic pursued by many authors [Keimel and Plotkin, 2016, Tix et al., 2009]. The technical difficulty in the definition of Plotkin powerdomain is the main problem in extending the theory.

Goncharov and Schröder [Goncharov and Schröder, 2011] rephrase powerdomains as large Lawvere theories, where Plotkin powerdomain monad is a simple example.

5.9.4 Comparison to other formalizations of category theory

Formalization of domain theory together with category theory is presented by Dockins [Dockins, 2014]. Formalization of category theory and monads based on the Univalent foundations appear in a series of papers by Arhens et al. [Ahrens et al., 2018, Ahrens et al., 2019b, Ahrens et al., 2019a].

Concrete categories are known to subsume small categories via Yoneda embedding. Certain large categories, for example homotopy categories that appear clasically in topology is known to be never concretizable [eter J. Freyd, 1970]. When concerning the applicability of our category theory framework, we believe this problem could be lifted by introducing larger universe of sets in an ad-hoc manner.

The idea of using categories as a package to handle functions with proofs was already presented by McBride [McBride, 1999, Chapter 7 and Section 3.1]. He also proposed the use of concrete categories for such a lightweight use of category theory, noting that the convertibility of terms is an easier way than propositional equality to handle the equational laws for morphisms, such as unit and associativity laws. His formal definition of categories differs from ours in that it also indexes on hom-sets, while in our definition, hom-sets are embedded as predicates. This difference further affects later definitions including that of functors. On the other hand, our definition makes it clearer that concrete categories are shallow embeddings of categories.

5.9.5 Category of convex spaces

Similarly to that the notion of convex spaces have been formulated many times in the literature 3.2, the category of convex spaces also have long been studied in the literature. Some early accounts are given by Swirszcz [Świrszcz, 1974] and Meng [Meng, 1987]. Recently there are applications of the category of convex spaces to abstract effects [Jacobs, 2010]. Although convex sets in vector spaces are intuitive objects, arguments on the category of convex spaces have some subtleties [Crhak, 2018a, Crhak, 2018b], indicating the need for formalizations.

Bibliography

[Aczel, 1977] Aczel, P. (1977). An introduction to inductive definitions**most of this paper was prepared while the author was visiting caltech. supported hy nsf grant 75-07562. In Barwise, J., editor, *Handbook of Mathematical Logic*, volume 90 of *Studies in Logic and the Foundations of Mathematics*, pages 739 – 782. Elsevier. (Cited on pages 27 and 28.)

[Affeldt et al., 2020a] Affeldt, R., Garrigue, J., Nowak, D., and Saikawa, T. (2020a). Monadic equational reasoning in Coq. `https://github.com/affeldt-aist/monae/`. Coq scripts. (Cited on pages 4, 51, 52, 53, 54, 55, 56, 61, 65, 66, 67, 69, and 70.)

[Affeldt et al., 2016] Affeldt, R., Garrigue, J., and Saikawa, T. (2016). Formalization of Reed-Solomon codes and progress report on formalization of LDPC codes. In *International Symposium on Information Theory and Its Applications (ISITA 2016), Monterey, California, USA, October 30–November 2, 2016*, pages 537–541. IEICE. IEEE Xplore. (Cited on page 1.)

[Affeldt et al., 2018] Affeldt, R., Garrigue, J., and Saikawa, T. (2018). Examples of formal proofs about data compression. In *International Symposium on Information Theory and Its Applications (ISITA 2018), Singapore, October 28–31, 2018*, pages 665–669. IEICE. IEEE Xplore. (Cited on pages 1 and 41.)

[Affeldt et al., 2020b] Affeldt, R., Garrigue, J., and Saikawa, T. (2020b). A library for formalization of linear error-correcting codes. *Journal of Automated Reasoning*. (Cited on page 1.)

[Affeldt et al., 2020c] Affeldt, R., Garrigue, J., and Saikawa, T. (2020c). Reasoning with conditional probabilities and joint distributions in Coq. *Computer Software*. Accepted by Computer Software, ISSN: 0289-6540. (Cited on page 1.)

[Affeldt et al., 2014] Affeldt, R., Hagiwara, M., and Sénizergues, J. (2014). Formalization of Shannon's theorems. *Journal of Automated Reasoning*, 53(1):63–103. (Cited on page 35.)

[Affeldt et al., 2019] Affeldt, R., Nowak, D., and Saikawa, T. (2019). A hierarchy of monadic effects for program verification using equational reasoning. In Hutton, G., editor, *Mathematics*

of Program Construction, pages 226–254, Cham. Springer International Publishing. (Cited on pages 1, 48, and 56.)

[Ahrens et al., 2019a] Ahrens, B., Frumin, D., Maggesi, M., and van der Weide, N. (2019a). Bicategories in Univalent Foundations. In Geuvers, H., editor, *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*, volume 131 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:17, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. (Cited on page 72.)

[Ahrens et al., 2018] Ahrens, B., Hirschowitz, A., Lafont, A., and Maggesi, M. (2018). High-Level Signatures and Initial Semantics. In Ghica, D. and Jung, A., editors, *27th EACSL Annual Conference on Computer Science Logic (CSL 2018)*, volume 119 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:22, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. (Cited on page 72.)

[Ahrens et al., 2019b] Ahrens, B., Hirschowitz, A., Lafont, A., and Maggesi, M. (2019b). Modular Specification of Monads Through Higher-Order Presentations. In Geuvers, H., editor, *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*, volume 131 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:19, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. (Cited on page 72.)

[Barendregt, 1991] Barendregt, H. (1991). Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154. (Cited on page 25.)

[Barr and Wells, 1985] Barr, M. and Wells, C. (1985). *Toposes, Triples and Theories*. Springer-Verlag, New York. (Cited on page 47.)

[Barras, 2010] Barras, B. (2010). Sets in coq, coq in sets. *Journal of Formalized Reasoning*, 3(1):29–48. (Cited on page 27.)

[Beaulieu, 2008] Beaulieu, G. (2008). *Probabilistic completion of nondeterministic models*. PhD thesis, University of Ottawa. (Cited on pages 68 and 69.)

[Bertot et al., 2008] Bertot, Y., Gonthier, G., Ould Biha, S., and Pasca, I. (2008). Canonical big operators. In Mohamed, O. A., Muñoz, C., and Tahar, S., editors, *Theorem Proving in Higher Order Logics*, pages 86–101, Berlin, Heidelberg. Springer Berlin Heidelberg. (Cited on page 39.)

[Bird, 1987] Bird, R. S. (1987). An introduction to the theory of lists. In Broy, M., editor, *Logic of Programming and Calculi of Discrete Design*, pages 5–42, Berlin, Heidelberg. Springer Berlin Heidelberg. (Cited on page 9.)

[Bird, 1989] Bird, R. S. (1989). Algebraic Identities for Program Calculation. *The Computer Journal*, 32(2):122–126. (Cited on page 9.)

[Bonchi et al., 2017] Bonchi, F., Silva, A., and Sokolova, A. (2017). The Power of Convex Algebras. In Meyer, R. and Nestmann, U., editors, *28th International Conference on Concurrency Theory (CONCUR 2017)*, volume 85 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 23:1–23:18, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. (Cited on pages 35, 36, and 71.)

[Cheung, 2017] Cheung, K.-H. (2017). *Distributive Interaction of Algebraic Effects*. PhD thesis, University of Oxford. (Cited on pages 63, 67, 68, and 71.)

[Church, 1932] Church, A. (1932). A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):346–366. (Cited on page 25.)

[Church, 1936] Church, A. (1936). An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363. (Cited on page 25.)

[Coquand and Huet, 1988] Coquand, T. and Huet, G. (1988). The calculus of constructions. *Information and Computation*, 76(2):95 – 120. (Cited on page 25.)

[Cover and Thomas, 2006] Cover, T. M. and Thomas, J. A. (2006). *Elements of information theory*. Wiley. 2nd ed. (Cited on pages 33 and 44.)

[Crhak, 2018a] Crhak, T. (2018a). A note on σ-algebras on sets of affine and measurable maps to the unit interval. (Cited on page 72.)

[Crhak, 2018b] Crhak, T. (2018b). On functors from category of giry algebras to category of convex spaces. (Cited on page 72.)

[Dockins, 2014] Dockins, R. (2014). Formalized, effective domain theory in coq. In Klein, G. and Gamboa, R., editors, *Interactive Theorem Proving*, pages 209–225, Cham. Springer International Publishing. (Cited on page 72.)

[eter J. Freyd, 1970] eter J. Freyd (1970). Homotopy is not concrete. In Peterson, F. P., editor, *The Steenrod Algebra and Its Applications: A Conference to Celebrate N.E. Steenrod's Sixtieth Birthday*, pages 25–34, Berlin, Heidelberg. Springer Berlin Heidelberg. (Cited on page 72.)

[Flood, 1981] Flood, J. (1981). Semiconvex geometry. *Journal of the Australian Mathematical Society*, 30(4):496–510. (Cited on pages 33, 37, 38, and 46.)

[Freyd, 1973] Freyd, P. J. (1973). Concreteness. *Journal of Pure and Applied Algebra*, 3(2):171 – 191. (Cited on page 65.)

[Fritz, 2009] Fritz, T. (2009). Convex spaces i: Definition and examples. (Cited on page 34.)

[Garillot et al., 2009] Garillot, F., Gonthier, G., Mahboubi, A., and Rideau, L. (2009). Packaging mathematical structures. In *TPHOLs*, volume 5674 of *Lecture Notes in Computer Science*, pages 327–342. Springer. (Cited on pages 29 and 34.)

[Gibbons, 2012] Gibbons, J. (2012). Unifying theories of programming with monads. In Wolff, B., Gaudel, M., and Feliachi, A., editors, *Unifying Theories of Programming, 4th International Symposium, UTP 2012, Paris, France, August 27-28, 2012, Revised Selected Papers*, volume 7681 of *Lecture Notes in Computer Science*, pages 23–67. Springer. (Cited on page 55.)

[Gibbons and Hinze, 2011] Gibbons, J. and Hinze, R. (2011). Just do it: simple monadic equational reasoning. In *16th ACM SIGPLAN Int. Conf. on Functional Programming (ICFP 2011), Tokyo, Japan, September 19–21, 2011*, pages 2–14. ACM. (Cited on pages 4, 10, 48, 50, 59, and 68.)

[Girard, 1972] Girard, J.-Y. (1972). *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris Diderot. (Cited on page 25.)

[Girard et al., 1989] Girard, J.-Y., Taylor, P., and Lafont, Y. (1989). *Proofs and Types*. Cambridge University Press, USA. (Cited on page 27.)

[Goncharov and Schröder, 2011] Goncharov, S. and Schröder, L. (2011). A counterexample to tensorability of effects. In Corradini, A., Klin, B., and Cîrstea, C., editors, *Algebra and Coalgebra in Computer Science*, pages 208–221, Berlin, Heidelberg. Springer Berlin Heidelberg. (Cited on pages 71 and 72.)

[Infotheo, 2020] Infotheo (2020). A Coq formalization of information theory and linear error-correcting codes. `https://github.com/affeldt-aist/infotheo/`. Coq scripts. (Cited on pages 34, 36, 38, 39, 40, 41, 42, 43, 44, 45, 66, 68, and 69.)

[Jacobs, 2010] Jacobs, B. (2010). Convexity, duality and effects. In *IFIP TCS*, volume 323 of *IFIP Advances in Information and Communication Technology*, pages 1–19. Springer. (Cited on pages 34, 36, and 72.)

[Jones and Plotkin, 1989] Jones, C. and Plotkin, G. D. (1989). A probabilistic powerdomain of evaluations. In *[1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science*, pages 186–195. (Cited on pages 46 and 71.)

[Keimel and Plotkin, 2016] Keimel, K. and Plotkin, G. (2016). Mixed powerdomains for probability and nondeterminism. *Logical Methods in Computer Science*, 13. (Cited on pages 33, 40, 46, and 72.)

[Keimel and Plotkin, 2009] Keimel, K. and Plotkin, G. D. (2009). Predicate transformers for extended probability and non-determinism. *Mathematical Structures in Computer Science*, 19(3):501–539. (Cited on page 46.)

[Kirch, 1993] Kirch, O. (1993). Bereiche und bewertungen. Master's thesis, Technischen Hochschule Darmstadt. (Cited on pages 33 and 46.)

[Kunen, 2009] Kunen, K. (2009). *The Foundations of Mathematics*, volume 19 of *Studies in Logic, Mathematical Logic and Foundations*. College Publications. (Cited on page 17.)

[Lee and Werner, 2011] Lee, G. and Werner, B. (2011). Proof-irrelevant model of cc with predicative induction and judgmental equality. *Logical Methods in Computer Science*, 7(4). (Cited on page 27.)

[Linton, 1966] Linton, F. E. J. (1966). Some aspects of equational categories. In Eilenberg, S., Harrison, D. K., MacLane, S., and Röhrl, H., editors, *Proceedings of the Conference on Categorical Algebra*, pages 84–94, Berlin, Heidelberg. Springer Berlin Heidelberg. (Cited on page 47.)

[Mac Lane, 1998] Mac Lane, S. (1998). *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer-Verlag, Berlin and New York, second edition. (Cited on page 48.)

[Mahboubi and Tassi, 2013] Mahboubi, A. and Tassi, E. (2013). Canonical structures for the working Coq user. In *4th Int. Conf. on Interactive Theorem Proving (ITP 2013), Rennes, France, July 22–26, 2013*, volume 7998 of *Lecture Notes in Computer Science*, pages 19–34. Springer. (Cited on page 29.)

[Martin-Löf, 1984] Martin-Löf, P. (1984). *Intuitionistic type theory*. Bibliopolis. (Cited on page 27.)

[McBride, 1999] McBride, C. (1999). *Dependently Typed Functional Programs and their Proofs*. PhD thesis, University of Edinburgh. (Cited on page 72.)

[Meng, 1987] Meng, X.-Q. (1987). *Categories of Convex Sets and of Metric Spaces with Applications to Stochastic Programming and Related Areas*. PhD thesis, State University of New York at Buffalo. (Cited on page 72.)

[Miquel and Werner, 2003] Miquel, A. and Werner, B. (2003). The not so simple proof-irrelevant model of cc. In Geuvers, H. and Wiedijk, F., editors, *Types for Proofs and Programs*, pages 240–258, Berlin, Heidelberg. Springer Berlin Heidelberg. (Cited on page 27.)

[Mislove, 2000] Mislove, M. (2000). Nondeterminism and probabilistic choice: Obeying the laws. In Palamidessi, C., editor, *CONCUR 2000 — Concurrency Theory*, pages 350–365, Berlin, Heidelberg. Springer Berlin Heidelberg. (Cited on page 71.)

[Mislove et al., 2004] Mislove, M., Ouaknine, J., and Worrell, J. (2004). Axioms for probability and nondeterminism. *Electronic Notes in Theoretical Computer Science*, 96:7 – 28. Proceedings of the 10th International Workshop on Expressiveness in Concurrency. (Cited on page 55.)

[Moggi, 1989] Moggi, E. (1989). Computational lambda-calculus and monads. In *LICS*, pages 14–23. IEEE Computer Society. (Cited on pages 10 and 47.)

[Morgan et al., 1996] Morgan, C., Mciver, A., Seidel, K., and Sanders, J. W. (1996). Refinement-oriented probability for csp. *Form. Asp. Comput.*, 8(6):617–647. (Cited on page 55.)

[Mu, 2019] Mu, S.-C. (2019). Equational reasoning for non-deterministic monad: A case study of Spark aggregation. Technical Report TR-IIS-19-002, Institute of Information Science, Academia Sinica. (Cited on pages 4 and 56.)

[Neumann, 1970] Neumann, W. D. (1970). On the quasivariety of convex subsets of affine spaces. *Archiv der Mathematik*, 21:11–16. (Cited on page 46.)

[Paulin-Mohring, 2015] Paulin-Mohring, C. (2015). Introduction to the Calculus of Inductive Constructions. In Paleo, B. W. and Delahaye, D., editors, *All about Proofs, Proofs for All*, volume 55 of *Studies in Logic (Mathematical logic and foundations)*. College Publications. (Cited on page 25.)

[Plotkin and Power, 2001] Plotkin, G. and Power, J. (2001). Semantics for algebraic operations. *Electronic Notes in Theoretical Computer Science*, 45:332 – 345. MFPS 2001,Seventeenth Conference on the Mathematical Foundations of Programming Semantics. (Cited on pages 10 and 48.)

[Plotkin, 1976] Plotkin, G. D. (1976). A powerdomain construction. *SIAM J. Comput.*, 5:452–487. (Cited on page 71.)

[Plotkin, 1983] Plotkin, G. D. (1983). Pisa notes on domain theory. (Cited on page 71.)

[Plotkin and Power, 2002] Plotkin, G. D. and Power, J. (2002). Notions of computation determine monads. In *FoSSaCS*, volume 2303 of *Lecture Notes in Computer Science*, pages 342–356. Springer. (Cited on page 48.)

[Reiterman, 1986] Reiterman, J. (1986). On locally small based algebraic theories. *Commentationes Mathematicae Universitatis Carolinae*, 27(2):325–340. (Cited on page 68.)

[Reynolds, 1984] Reynolds, J. C. (1984). Polymorphism is not set-theoretic. In Kahn, G., MacQueen, D. B., and Plotkin, G., editors, *Semantics of Data Types*, pages 145–156, Berlin, Heidelberg. Springer Berlin Heidelberg. (Cited on page 27.)

[Semadini, 1971] Semadini, Z. (1971). *Banach Spaces of Continuous Functions*. PWN. (Cited on page 46.)

[Shoenfield, 1967] Shoenfield, J. R. (1967). *Mathematical Logic*. Addison-Wesley. (Cited on page 17.)

[Stone, 1949] Stone, M. H. (1949). Postulates for the barycentric calculus. *Ann. Mat. Pura Appl.*, 29(1):25–30. (Cited on pages 33, 34, 36, 37, and 39.)

[Świrszcz, 1974] Świrszcz, T. (1974). Monadic functors and convexity. *Bulletin de l'Académie polonaise des sciences. Série des sciences mathématiques, astronomiques et physiques*, 22(1). (Cited on pages 34 and 72.)

[Tix et al., 2009] Tix, R., Keimel, K., and Plotkin, G. (2009). Semantic domains for combining probability and non-determinism. *Electronic Notes in Theoretical Computer Science*, 222:3 – 99. (Cited on pages 46 and 72.)

[van Heerdt et al., 2018] van Heerdt, G., Hsu, J., Ouaknine, J., and Silva, A. (2018). Convex language semantics for nondeterministic probabilistic automata. In Fischer, B. and Uustalu, T., editors, *Theoretical Aspects of Computing – ICTAC 2018*, pages 472–492, Cham. Springer International Publishing. (Cited on pages 34, 35, and 71.)

[Varacca and Winskel, 2006] Varacca, D. and Winskel, G. (2006). Distributing probability over non-determinism. *Mathematical Structures in Computer Science*, 16(1):87–113. (Cited on pages 33, 37, 46, 63, 69, and 71.)

[Wadler, 1990] Wadler, P. (1990). Comprehending monads. In *LISP and Functional Programming*, pages 61–78. (Cited on pages 10 and 48.)

[Werner, 1997] Werner, B. (1997). Sets in types, types in sets. In Abadi, M. and Ito, T., editors, *Theoretical Aspects of Computer Software*, pages 530–546, Berlin, Heidelberg. Springer Berlin Heidelberg. (Cited on page 27.)

[Williams, 1969] Williams, N. H. (1969). On grothendieck universes. *Compositio Mathematica*, 21(1):1–3. (Cited on page 28.)

[Winskel, 1983] Winskel, G. (1983). A note on powerdomains and modality. In Karpinski, M., editor, *Foundations of Computation Theory*, pages 505–514, Berlin, Heidelberg. Springer Berlin Heidelberg. (Cited on page 72.)

[Yi and Larsen, 1992] Yi, W. and Larsen, K. G. (1992). Testing probabilistic and nondeterministic processes. In LINN, R. and UYAR, M., editors, *Protocol Specification, Testing and Verification, XII*, IFIP Transactions C: Communication Systems, pages 47 – 61. Elsevier, Amsterdam. (Cited on page 55.)