A Monad For Randomized Algorithms

an abstract
submitted on the fifteenth day of april, 2016
to the department of mathematics
of the school of science and engineering of
tulane university
in partial fulfillment of the requirements
for the degree of
doctor of philosophy
by

Tyler C. Barker

approved:

Michael Mislove, ph.d.
chairman

Rafal Komendarczyk, ph.d.

Slawomir Kwasik, ph.d.

Ramgopal Mettu, ph.d.

Albert Vitter, ph.d.

Abstract

This thesis presents new domain-theoretic models for randomized algorithms. A randomized algorithm can use random bits from an oracle to control its computation. The possible random bits form a binary tree, where each random choice of a bit is a branching of the tree. The randomized algorithm then determines what the output should be for each branch. This idea forms the basis of our random choice functors. However, the functor only provides one half of the model. We must also show how multiple randomized algorithms can be combined or composed. This is where the monadic structure comes into play. If we wish to join multiple randomized algorithms to form one resulting algorithm, then we can run each algorithm in parallel, using the same random bits for each.

Monads are used to add a computational effect to an existing semantic model. In order to work with models of the lambda calculus, it is important to work in a Cartesian closed category of domains, due to Lambek's theorem and Scott's corollary. Our first random choice monad is shown to be an endofunctor of the Cartesian closed category BCD. If we wish to add multiple computational effects, then we can compose monads as long as the monads enjoy a distributive law. It is shown that in the category BCD, our first random choice monad enjoys a distributive law with the lower powerdomain for nondeterminism. Two variations of the random choice monad are then given. The first variation has a distributive law with the convex powerdomain in the categories RB and FS, while the second variation has a distributive law with the upper powerdomain in BCD.

We use the random choice monads to develop a new programming language, Randomized PCF. This extends the language PCF by adding in random choice, allowing for the programming of randomized algorithms. A full operational semantics is given for Randomized PCF, and a random choice monad is used to give it a mathematical model (denotational semantics). Finally, an implementation of Randomized PCF is developed, and the Miller-Rabin algorithm is implemented in Randomized PCF.

A Monad For Randomized Algorithms

A DISSERTATION
SUBMITTED ON THE FIFTEENTH DAY OF APRIL, 2016
TO THE DEPARTMENT OF MATHEMATICS
OF THE SCHOOL OF SCIENCE AND ENGINEERING OF
TULANE UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
BY

Tyler C. Barker

APPROVED:

Michael Mislove, ph.d.
CHAIRMAN

Rafal Komendarczyk, ph.d.

Slawomir Kwasik, ph.d.

Ramgopal Mettu, ph.d.

Albert Vitter, ph.d.

Acknowledgments

I would first and foremost like to thank my thesis advisor, Michael Mislove. His guidance was instrumental in shaping my research and the writing of this thesis. My time as a graduate student was a tremendous experience. Mike gave me the freedom to direct my own research, which was difficult at times. It was a struggle to find a research problem, but it made it much more rewarding in the end.

At Tulane, I was able to travel around the world to conferences and workshops, meeting the foremost experts in my field. I would like to thank everyone that I have met who has expressed interest in my work. I also would like to thank all of the mathematicians and computer scientists whose work preceded my own. I especially thank Dana Scott and Jean Goubault-Larrecq. Their work provided the foundation for this thesis, and they have been graciously willing to discuss ideas with me.

Finally, I would like to thank my parents, Gary and Carol, and my brothers, Travis and Todd, for their continued love and support throughout this entire process. Growing up with my family led me to where I am today. I do not yet know what the future holds for me, but I know that my family will be there to support me at every step.

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Background

1.1 Introduction

Starting with Dana Scott's model of the untyped lambda calculus, domain theory has been largely successful in providing models of computation. Domain theory lends itself to programming language semantics due to its wealth of Cartesian closed categories that can be used as semantic models. The use of domain theory has expanded to provide denotational semantics for many computational effects, such as continuation and nondeterminism, using Moggi's [1] monadic approach. One type of computation that has been problematic to model, however, is probabilistic computation. The most well known monad of probabilistic computation is the probabilistic powerdomain, first defined by Saheb-Djahromi in 1980 [2]. However, this monad has two major flaws [3]. First, it is not known whether any Cartesian closed category of domains is closed under the probabilistic powerdomain. There has been no real progress on this frustrating question in the more than 35 years since Saheb-Djahromi's initial work. The category of coherent domains is closed under this construction, but it is not Cartesian closed. Second, there is no distributive law between the probabilistic powerdomain and any of the three nondeterministic powerdomains [4]. According to Beck's Theorem [5], the composition of two monads is a monad if and only if the monads satisfy a distributive law. Thus, to generate a monad from the probabilistic powerdomain and any of the monads for nondeterministic choice, new laws must be added, an approach explored independently by Tix [6, 7] and Mislove [8].

To address these flaws, work has been done to develop alternate models of probabilistic computation. Varacca and Winskel [4, 9] constructed what they called *indexed valuation monads*. These monads leave various Cartesian closed categories of domains invariant, at the expense of abandoning the laws of the probability monad. Indexed valuations weaken the laws of probabilistic choice, no longer requiring that $p +_r p = p$, where $p +_r q$ denotes choosing p with probability r and q with probability $1 - r$. In this setting, it is possible to satisfy a distributive law with the nondeterministic powerdomains.

Mislove [10] built upon this work, using an indexed valuation model to define a monad of finite random variables. The Cartesian closed categories RB and FS were shown to be closed under this construction. Later, Goubault-Larrecq and Varacca [11] proposed a model of continuous random variables over the Cartesian closed category BCD, but the model did not form a monad in this category [12]. The models that this thesis describes are based upon these continuous random variables, in particular, the uniform continuous random variables.

This thesis presents new domain-theoretic models for randomized algorithms. Given some source of randomness (an oracle), a randomized algorithm can use random bits from the oracle to control its computation. The possible random bits form a binary tree, where each random choice of a bit is represented by a branching of the tree. The randomized algorithm then determines what the output should be for each branch. This idea forms the basis of our random choice functors. However, the functor only provides one half of the model. We must also show how multiple randomized algorithms can be combined or composed. This is where the monadic structure comes into play. If we wish to join multiple randomized algorithms to form one resulting algorithm, then we can run each algorithm in parallel, using the same random bits for each.

Monads are used to add a computational effect to an existing semantic model. In order to work with models of the lambda calculus, it is important to work in a Cartesian closed category of domains, due to Lambek's theorem [13]. Lambek showed that there is a one-to-one correspondence between models of the typed lambda calculus and Cartesian closed categories. Scott's corollary [14] states that models of the untyped lambda calculus all arise as reflexive objects in Cartesian closed categories. The first random choice monad presented in this thesis is shown to be an endofunctor of the Cartesian closed category BCD. It was a subcategory of BCD that Scott used in his first model of the untyped lambda calculus.

If we wish to add multiple computational effects, then we can compose monads. However, this composition forms a monad only if the monads enjoy a distributive law. It is shown that in the category BCD, our first random choice monad enjoys a distributive law with the lower powerdomain for nondeterminism. A second random choice monad, which differs slightly from the first, is then presented as an endofunctor in the categories RB and FS. In these categories, we display a distributive law with this monad and the convex powerdomain for nondeterminism. Finally, a third random choice monad is given that enjoys a distributive law with the upper powerdomain in the category BCD.

We use the random choice monads to develop a new programming language, Randomized PCF. This extends the language PCF by adding in random choice, allowing for the programming of randomized algorithms. PCF is the toy language that Scott used to illustrate how his model of the lambda calculus could be applied to model programming languages; it is a focus of study for semantics. A full operational semantics is given for Randomized PCF, and a random choice monad is used to give it a mathematical model (denotational semantics).

The final chapter of the thesis describes how the random choice monads can be implemented

in functional programming languages. Versions of the monads are given in the programming languages Haskell and Scala, and a formal proof of the monad laws is given using the interactive theorem prover Isabelle. Finally, an implementation of Randomized PCF is developed. The grammar for the syntax is designed, and a parser and interpreter are written in SML. As a proof of concept, a program implementing the Miller-Rabin algorithm in Randomized PCF is displayed.

Structure of the Thesis

Chapter 1 contains the relevant background information needed for the rest of the thesis. Chapter 2 defines the first random choice functor and proves that it satisfies the monad laws when viewed as an endofunctor of the category BCD. Motivation is given for the functor and the Kleisli extension of the monad. Chapter 3 first gives a distributive law between the random choice monad and the lower powerdomain. Then, variations of the monad are defined, along with distributive laws with the convex and upper powerdomains. Chapter 4 gives an operational and denotational semantics for Randomized PCF, along with proofs that the semantics coincide. Chapter 5 discusses how the random choice monads can be implemented in functional programming. Monad definitions are given in Haskell and Scala, and a formal proof of the monad laws is given in Isabelle. Finally, an implementation of Randomized PCF is given, along with some example programs written in the language.

1.2 Background

1.2.1 Lambda Calculus

The lambda calculus was invented by Alonzo Church around 1928, and first published in 1932 [15]. A more extensive history of the lambda calculus can be found in [16]. A comprehensive treatise on the lambda calculus was written by Barendregt [17].

The terms of the lambda calculus are given by the following Backus-Naur Form (BNF):

$$M, N \ ::= \quad x \ \mid \ (MN) \ \mid \ (\lambda x.M)$$

The lambda calculus is built from an infinite set of variables along with function abstraction and application. If x is a variable, then x is a term of the lambda calculus. If M and N are both terms, then so is MN (application). Finally, if M is a term and x is a variable, then $\lambda x.M$ is a term (function abstraction). This is called the untyped lambda calculus, since all terms are at the same level. This means that each term is a function taking other terms as input, as well as an argument

for any other terms as a function.

Example 1.1. The term $\lambda x.x$ is the identity function. Given any x, it outputs x.

The lambda abstraction $\lambda x.M$ *bounds* the variable x. Variables not bound are said to be *free*.

Definition 1.2. The set of free variables of a term M, $\mathrm{FV}(M)$, is defined inductively:

$$\mathrm{FV}(x) = x$$
$$\mathrm{FV}(MN) = \mathrm{FV}(M) \cup \mathrm{FV}(N)$$
$$\mathrm{FV}(\lambda x.M) = \mathrm{FV}(M) - \{x\}$$

A term is closed if it has no free variables. An important aspect of the lambda calculus is the notion of substituting some term N for all free occurrences of a variable x in another term M. This is denoted $M[N/x]$ here, and is defined as follows:

$$x[N/x] = N$$
$$y[N/x] = y, \text{if } x \neq y$$
$$(M_1 M_2)[N/x] = (M_1[N/x])(M_2[N/x])$$
$$(\lambda y.M)[N/x] = \lambda y.(M[N/x])$$

There are two concerns that one should have when performing a substitution. First, as stated, only free occurrences of a variable should be replaced. For example, $x(\lambda xy.x)[N/x]$ does not equal $N(\lambda xy.N)$. The x in the lambda term is bound, so it should not get replaced. Instead, $x(\lambda xy.x)[N/x] = N(\lambda xy.x)$.

The second concern of substitution is avoiding the capture of free variables by ones that are bound. For example, let $N \equiv \lambda z.xz$. Now consider $(\lambda x.yx)[N/y]$. Note that x is bound in $\lambda x.yx$ but free in N. If we perform the substitution, we get $\lambda x.(\lambda z.xz)x$. Here the free x in N gets captured. This mistakenly treats the x in both terms as the same variable. To get around this, we should rename the bound variable before performing the substitution. We can change $\lambda x.yx$ to $\lambda w.yw$. Then the substitution results in $\lambda w.(\lambda z.xz)w$.

As a theory, terms of the lambda calculus are distinct unless they are identical. We now add some conversion rules that allow terms to be identified - this process of applying the rules is

called reduction. Terms of the lambda calculus can be converted and reduced in three main ways: α-equivalence, β-reduction, and η-conversion.

First, *α-equivalence* allows for the renaming of bound variables. For example, $\lambda x.x$ is equivalent to $\lambda y.y$. The rule for α-equivalence can be stated by:

$$\lambda x.M \equiv_\alpha \lambda y.(M[y/x])$$

This was used above in avoiding the capture of free variables by substitution. Care must be taken to ensure that the new variable name is not captured by some other abstraction. For example, in the term $\lambda x.\lambda y.x$, x could not be renamed to y. Having infinitely many variables means there is always a "fresh" variable to use.

Second, *β-reduction* is how a function gets applied to its arguments. This is shown in the following rule:

$$(\lambda x.M)N \to_\beta M[N/x]$$

For example, $(\lambda x.x)y \to_\beta x[y/x] = y$, which shows that $\lambda x.x$ is indeed the identity function.

Finally, *η-conversion* says that a function can be fully defined by what it outputs for all possible arguments.

$$\lambda x.(fx) \equiv_\eta f, x \notin \mathrm{FV}(f)$$

In the untyped lambda calculus, the reduction of a term does not necessarily terminate. An example is the term $(\lambda x.xx)(\lambda x.xx)$. Using β-reduction, $(\lambda x.xx)(\lambda x.xx) \to_\beta xx[(\lambda x.xx)/x] = (x[(\lambda x.xx)/x])(x[(\lambda x.xx)/x]) = (\lambda x.xx)(\lambda x.xx)$.

Simply Typed Lambda Calculus

Types can be added to the lambda calculus to form a simply typed lambda calculus. We assume that there is some set of ground types. If ι is a ground type, we give the set of simple types in the following BNF:

$$s, t \ ::= \ \iota \mid s \to t \mid s \times t \mid ()$$

The type $s \to t$ is the type of functions from s to t, and $s \times t$ is the type of pairs (x, y), where x is of type s and y is of type t. The type $()$ is the unit type, which has only one element, $*$.

The terms of the simply typed lambda calculus are now given by the following BNF:

$$M, N \ ::= \ x \mid (MN) \mid (\lambda x.M) \mid (M, N) \mid \pi_1(M) \mid \pi_2(M) \mid *$$

[Var]	$H, x : t \vdash x : t$
[App]	$\dfrac{H \vdash M : s \to t \quad H \vdash N : s}{H \vdash M(N) : t}$
[Lambda]	$\dfrac{H, x : s \vdash M : t}{H \vdash \lambda x.M : s \to t}$
[Pair]	$\dfrac{H \vdash M : s \quad H \vdash N : t}{H \vdash (M, N) : s \times t}$
$[\pi_1]$	$\dfrac{H \vdash M : s \times t}{H \vdash \pi_1(M) : s}$
$[\pi_2]$	$\dfrac{H \vdash M : s \times t}{H \vdash \pi_2(M) : t}$
$[*]$	$H \vdash * : ()$

Table 1.1: Typing Rules for the Simply Typed Lambda Calculus

(M, N) is a pair of terms, and π_1 and π_2 project a pair to each of its components. Thus, $\pi_1(M, N) = M$ and $\pi_2(M, N) = N$. Now that each term has a type, there are limitations to how terms can be form and combined. The typing rules are displayed in Table 1.1. The notation $M : t$ means that M is of type t. Here, H is a type assignment, which is a list of variables and their types. The statement $H \vdash M : t$ means that under type assignment H, the term M has type t.

1.2.2 Domain Theory

The lambda calculus just described is a theory - to be meaningful, it also needs a model. It was not until the late 1960s that the first model of the untyped lambda calculus was found. It was the work of Dana Scott [18] and the model he devised that founded the area of domain theory.

Domain theory is a branch of mathematics used in specifying a *denotational semantics* for a programming language. This involves constructing mathematical objects that can describe the meaning of a programming language's expressions. For a more thorough introduction to domain theory, consult [19, 20].

Domain theory involves working with partially ordered sets, or *posets*.

Definition 1.3. A *poset* is a set P with a binary relation, $\sqsubseteq$, such that for all $x, y, z \in P$:

1. [Reflexivity] $x \sqsubseteq x$
2. [Transitivity] $x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$
3. [Antisymmetry] $x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$

Example 1.4. Any total order, such as real numbers with the usual ordering, forms a poset.

Example 1.5. A family of sets ordered by inclusion ($M \sqsubseteq N \Leftrightarrow M \subseteq N$) form a poset.

When considering functions between posets, it is usually desired that the functions preserve the partial order. Such functions are called monotone.

Definition 1.6. For two posets, P and Q, the function, $f : P \to Q$ is *monotone* if for all x, y in P, $x \sqsubseteq y$ implies $f(x) \sqsubseteq f(y)$ in Q.

Proposition 1.7. *The set of monotone functions between two posets, P and Q, forms another poset in the pointwise order: $f \sqsubseteq g$ if for all $x \in P$, $f(x) \sqsubseteq g(x)$ in Q.*

It is often useful to think of elements of a poset as possible stages of a computation. As a program is being executed, each stage of computation moves up in the order, getting closer and closer to the desired output, which may be the least upper bound, or supremum, of the partial computations. However, there may be multiple ways to get to the same desired output, and elements of distinct paths may not compare. An example given in [21] is of a computer solving a crossword puzzle. At first, the puzzle is empty, and this stage of computation is the bottom element. A complete puzzle represents the top element, but there are many ways to move from the bottom to the top. We can order the partially completed puzzles by $x \sqsubseteq y$ if any word completed in x is also completed in y. Two elements x and y may not compare, but we can always find an element above both of them by filling in all words completed in either x or y. This motivates the following definition:

Definition 1.8. A nonempty subset, D, of a poset P is *directed* if for any two elements in D, x and y, there is an upper bound z also in D ($x \sqsubseteq z$ and $y \sqsubseteq z$).

Example 1.9. Any chain (a subset in which any pair of elements is comparable) is directed. For any pair, the larger of the two is an upper bound.

Definition 1.10. A poset in which every directed subset D has a least upper bound (denoted $\bigsqcup D$) is called a *directed-complete partial order*, or *dcpo* for short.

Example 1.11. Any finite poset is a dcpo. A closed interval of real numbers (with the usual ordering) is a dcpo. However, the natural numbers with their usual order do not form a dcpo. The natural numbers themselves form a directed family with no (least) upper bound.

When considering functions between two dcpos, it is usually desired that monotone functions also preserved directed suprema (least upper bounds).

Definition 1.12. For two dcpos P and Q, a monotone function, $f : P \to Q$ is *Scott continuous* if for any directed family $D \subseteq P$, $f(\bigsqcup D) = \bigsqcup f(D)$

The set of Scott continuous functions between two dcpos P and Q, denoted $[P \to Q]$, also forms a dcpo using the same pointwise order described above for monotone functions. The proof is a simple exercise.

The following is the least fixed-point theorem for Scott continuous functions (sometimes called the Kleene fixed-point theorem). This theorem is crucial in denotational semantics to give meaning to recursive functions in programming languages, as is discussed in Section 2.1.

Theorem 1.13. *Let D be a dcpo with a least element $\bot$. Then every Scott continuous self-map $f : D \to D$ has a least fixed-point. It is given by $\bigsqcup_{n\in\mathbb{N}} f^n(\bot)$.*

Proof. Consider the set $\{f^n(\bot) \mid n \in \mathbb{N}\}$. This forms a chain. $\bot \sqsubseteq f(\bot)$, so by monotonicity of f, $f(\bot) \sqsubseteq f^2(\bot)$ and so on for each $f^n(\bot)$. Since D is a dcpo, this chain (and thus, directed subset) has a supremum, $\bigsqcup_{n\in\mathbb{N}} f^n(\bot)$, and since f is Scott continuous,

$$f(\bigsqcup_{n\in\mathbb{N}} f^n(\bot)) = \bigsqcup_{n\in\mathbb{N}} f^{n+1}(\bot) = \bigsqcup_{n\in\mathbb{N}} f^n(\bot)$$

so $\bigsqcup_{n\in\mathbb{N}} f^n(\bot)$ is a fixed point of f.

To show this is the least fixed point, assume there is another fixed point x. Clearly, $\bot \sqsubseteq x$, and by monotonicity of f, $f(\bot) \sqsubseteq f(x) = x$. Repeatedly applying f gives that $f^n(\bot) \sqsubseteq x$ for all n. Thus, x is an upper bound for $\{f^n(\bot) \mid n \in \mathbb{N}\}$, and since $\sqcup_{n\in\mathbb{N}} f^n(\bot)$ is the least upper bound, it must be below x. ■

It was explained above how a total computation can be viewed as a directed supremum of partial computations. However, it may not be possible to complete the total computation, like calculating the entire decimal expansion of a real number. In this case, we would like to approximate the total computation with a much simpler partial computation that we can compute. Then we would like the total computation to be the supremum of these simpler approximations. This notion is made precise below.

Definition 1.14. If D is a dcpo and $x, y \in D$, then x *approximates* y (denoted $x \ll y$) iff for every directed set S with $y \sqsubseteq \bigsqcup S$, there is some $s \in S$ such that $x \sqsubseteq s$. We also say that x is way below y if $x \ll y$. We denote $\downarrow y = \{x \in D | x \ll y\}$, and similarly, $\uparrow x = \{y \in D | x \ll y\}$.

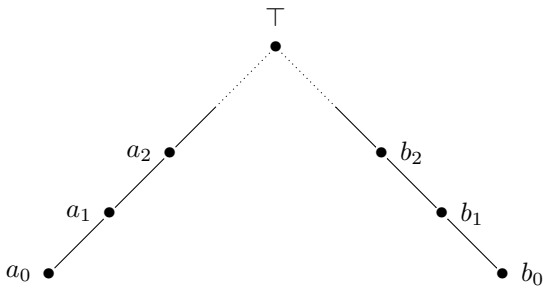


Figure 1.1: A dcpo that is not a domain

Definition 1.15. An element x of a dcpo D is *compact* (or *finite*) if it approximates itself ($x \ll x$). The set of compact elements of D is denoted $\mathsf{K}(D)$.

Proposition 1.16. *Let w, x, y, z be elements of a dcpo D.*

1. $x \ll y \Rightarrow x \sqsubseteq y$
2. $w \sqsubseteq x \ll y \sqsubseteq z \Rightarrow w \ll z$

Definition 1.17. A dcpo D is a *domain* if for all d in D, $\downarrow d$ is directed and $\bigsqcup \downarrow d = d$.

Example 1.18. Figure 1.1 displays an example of a dcpo that is not a domain. There are two infinite chains, $\{a_i\}$ and $\{b_j\}$, each with supremum $\top$, but no a_i and b_j are comparable. Note that for any i, a_i is not way below $\top$ because $\{b_j\}$ is a directed family whose supremum is $\top$, but no element b_j is above a_i. Similarly, for any j, b_j is not way below $\top$. Thus, the set of elements way below $\top$ is empty, and therefore, not directed.

Definition 1.19. A dcpo D is *algebraic* if for all d in D, the set of compact elements below d, $\downarrow d \cap \mathsf{K}(D)$ is directed and $\bigsqcup(\downarrow d \cap \mathsf{K}(D)) = d$.

Note that any algebraic dcpo is necessarily a domain.

Example 1.20. Suppose $D = \mathcal{P}(X)$, the power set of some set X, ordered by inclusion. The compact elements of D are precisely the finite subsets of X. Every set is the union (supremum) of its finite subsets, and the finite subsets of a set is directed, so D is algebraic.

Example 1.21. The set of real numbers, $\mathbb{R}$, with the usual order forms a domain ($x \ll y$ if and only if $x < y$). However, there are no compact elements, so it is not algebraic.

Theorem 1.22. *If D is a domain, and $x \ll y$, then $x \ll z \ll y$ for some z in D.*

Example 1.23. Consider the set of intervals, $\{[a,b] \mid a,b \in \mathbb{R}, a \leq b\}$, ordered by reverse inclusion. For any directed subset D, any two elements have a nonempty intersection, and $\bigsqcup D = \bigcap_{d \in D} d$. The maximal elements are the degenerate intervals, $[x,x]$. The way below relation can be characterized by $[a,b] \ll [c,d]$ if $[c,d] \subseteq (a,b)$. It can easily be shown that this forms a domain. This is called the *interval domain*.

We can view this domain as partial computations in search for some real number x. We may never compute x exactly, but at each stage of computation, we have interval containing x that gets smaller and smaller. Thus, ordering the intervals by reverse inclusion is really ordering by the information we have about x.

Topology

Definition 1.24. Let X be a set. A *topology* on X is a collection of subsets of X, called *open sets*, such that every union of open sets is open, and every finite intersection of opens sets is open. Note that viewed as the empty union and empty intersection, respectively, both the empty set, $\emptyset$, and the whole set, X, must be open.

Definition 1.25. A *closed set* of a topological space X is the complement, $X \setminus U$, of some open subset, U, of X. For any set F, the *closure* of F, denoted $\overline{F}$, is the smallest closed set containing F.

Definition 1.26. A function $f : X \to Y$ between two topological spaces in *continuous* if for any open set $V \subseteq Y$, the inverse image, $f^{-1}(V) = \{x \in X \mid f(x) \in V\}$ is an open set in X.

Definition 1.27. Let X be a topological space, and let $\mathcal{B}$ be a family of open sets in X. Then $\mathcal{B}$ forms a *basis* for the topology on X if and only if every open set is a union of elements in $\mathcal{B}$.

For a dcpo D, we can define a topology, called the Scott topology, on D. Then many of the above properties can be given topological characterizations.

Definition 1.28. For a poset D, a set U is a *lower set* if $x \in U$ and $y \sqsubseteq x \Rightarrow y \in U$. Similarly, U is an *upper set* if $x \in U$ and $x \sqsubseteq y \Rightarrow y \in U$. Also, for an arbitrary set $U \subseteq D$, the *lower set* of U, denoted $\downarrow U$, is equal to $\{d \in D \mid \exists u \in U, d \sqsubseteq u\}$. Similarly, for the *upper set* of U, $\uparrow U = \{d \in D \mid \exists u \in U, u \sqsubseteq d\}$. If U is a lower set, then $U = \downarrow U$, and if U is an upper set, $U = \uparrow U$.

Definition 1.29. A subset U of a dcpo D is *Scott open* if it is an upper set and if for any directed family S with $\bigsqcup S \in U$, there exists $s \in S$ such that $s \in U$.

A set is *Scott closed* if is a lower set and closed under directed suprema. For any element x, the closure of the singleton $\{x\}$ is simply its lower set, $\downarrow x$.

Proposition 1.30. *Suppose X and Y are two posets with the Scott topology. A function $f : X \to Y$ is continuous in the Scott topology if and only if the f is Scott continuous.*

Proposition 1.31. *If D is a domain, then $\Uparrow x$ is Scott open for all $x \in D$. In fact, these open sets form a basis for the Scott topology on D. Each open set, U, of D can be written as $\bigcup_{x \in U} \Uparrow x$.*

Proposition 1.32. *If D is algebraic, then the open sets, $\uparrow x$, $x \in \mathsf{K}(D)$, form a basis for the Scott topology on D. Each open set, U, of D can be written as $\bigcup_{x \in U \cap \mathsf{K}(D)} \uparrow x$.*

Definition 1.33. A subset S of a topological space X is *saturated* if it is the intersection of the open sets that contain it.

For a poset with the Scott topology, saturated sets are simply the upper sets.

Definition 1.34. A domain D is *coherent* if the intersection of any two compact saturated sets is D is again compact.

We now describe another topology that can be placed on a dcpo.

Definition 1.35. For a dcpo D, the *Lawson topology* is the smallest topology containing the Scott-open sets and sets of the form $D \setminus \uparrow x, x \in D$.

Definition 1.36. A topology on a space X is *Hausdorff* if for every pair, x and y, of distinct points in X, there are open sets, U and V, such that $x \in U, y \in V$, and $U \cap V = \emptyset$.

Proposition 1.37. *If D is a domain, then the Lawson topology on D is Hausdorff.*

Definition 1.38. Suppose K is a subset of a topological space X. An *open cover* of K is family of open sets, $\{V_i\}$ such that $K \subseteq \bigcup_i V_i$. K is *compact* if and only if every open cover of K has a finite subcover.

Proposition 1.39. *A domain D is coherent if and only if it is compact in the Lawson topology.*

1.2.3 Category Theory

Category theory is a formalism of mathematical structures and transformations between them. Developed by Samuel Eilenberg and Saunders Mac Lane in the early 1940s, category theory was first used as a bridge between abstract algebra and topology. It has since been applied to many

other areas of mathematics and beyond, such as logic, computer science, and philosophy. For a more thorough introduction to category theory, one can consult the classic book by Mac Lane [22] or the more recent text by Awodey [23].

Definition 1.40. A *category* $\mathcal{C}$ consists of a collection of objects, $\mathsf{Ob}(\mathcal{C})$, and a collection of morphisms, $\mathsf{Hom}(\mathcal{C})$. A morphism f has a domain, A, and a codomain, B, which are each objects. This is denoted $f : A \to B$. For any two objects A and B, the *hom-set* is defined as

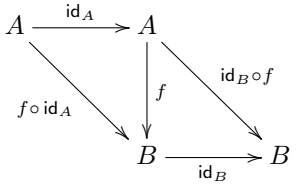
$$\mathcal{C}(A, B) \equiv \{f \in \mathsf{Hom}(\mathcal{C}) \mid f : A \to B\}$$

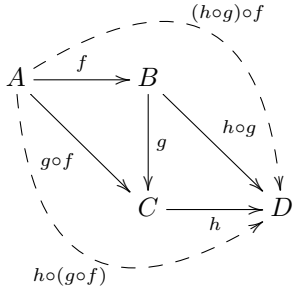
For any three objects, A, B, and C, there is a binary operation, called composition, of the form $\mathcal{C}(A, B) \times \mathcal{C}(B, C) \to \mathcal{C}(A, C)$. For two morphisms, $f : A \to B$ and $g : B \to C$, the composition of f and g is denoted $g \circ f : A \to C$. For every object A there is an *identity* morphism, $\mathsf{id}_A : A \to A$. If $f : A \to B$, $g : B \to C$, and $h : C \to D$, the following equations must hold:

$$f \circ \mathsf{id}_A = f = \mathsf{id}_B \circ f$$

$$h \circ (g \circ f) = (h \circ g) \circ f$$

In category theory, many definitions require some set of equations to hold. It often is easier to understand these equations using commutative diagrams. For example, we can express the above equations as the following diagrams:



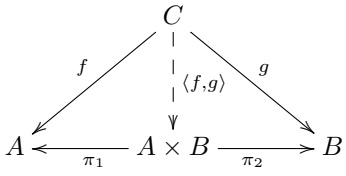


Example 1.41. The most basic category is Set, whose objects are sets and whose morphisms are functions between sets. Composition is the usual function composition.

Definition 1.42. An object I in a category $\mathcal{C}$ is *initial* if for every object A, there is a unique morphism from I to A. An object T is *terminal* if for every object A, there is a unique morphism from A to T.

Example 1.43. In Set, the initial object is the empty set and any singleton set is a terminal object.

Definition 1.44. If A and B are objects in a category $\mathcal{C}$, then a *product* of A and B is an object $A \times B$ along with pair of projections, $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$, such that for any object C and morphisms $f : C \to A$ and $g : C \to B$, there is a unique morphism $\langle f, g \rangle : C \to A \times B$ where the following diagram commutes.



The definition above defines a binary product of two objects. This can be extended to define a product an arbitrary family of objects.

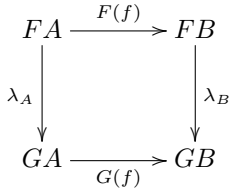
Example 1.45. In the category Set, the categorical product is simply the Cartesian product.

Definition 1.46. For two categories $\mathcal{C}$ and $\mathcal{D}$, a *functor* $F : \mathcal{C} \to \mathcal{D}$ consists of an object map and a morphism map. The object map assigns to every object A of $\mathcal{C}$ an object FA of $\mathcal{D}$. For a morphism $f : A \to B$, the morphism map gives a morphism $F(f) : FA \to FB$ such that the following equations hold:

$$F(\mathsf{id}_A) = \mathsf{id}_{FA}$$

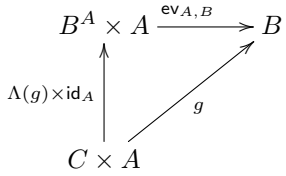
$$F(g \circ f) = F(g) \circ F(f)$$

Definition 1.47. If $F, G : \mathcal{C} \to \mathcal{D}$ are two functors, then a *natural transformation* $\lambda : F \to G$ associates to each object A of $\mathcal{C}$ a morphism, $\lambda_A : FA \to GA$, in $\mathcal{D}$, such that for any morphism $f : A \to B$, the following diagram commutes:



Definition 1.48. A category $\mathcal{C}$ that has binary products also has *exponentials* if for all objects A and B, there is an object B^A of $\mathcal{C}$ and a morphism, $\mathsf{ev}_{A,B} : B^A \times A \to B$, such that for any morphism $g : C \times A \to B$ there is a unique morphism $\Lambda(g) : C \to B^A$ such that the following

diagram commutes:



Example 1.49. In the category Set, for two sets A and B, the exponential object B^A is the set of all functions from $A \to B$. The morphism $\mathsf{ev}_{A,B}$ is just the evaluation map, and for a map $g : C \times A \to B$, the map $\Lambda(g) : C \to B^A$ is the curried version of g:

$$\Lambda(g)(c)(a) = g(c, a)$$

Cartesian Closed Categories

In this thesis, we work within Cartesian closed categories as these are necessary to model lambda calculi [24].

Definition 1.50. A category is a *Cartesian closed category (CCC)* if is has a terminal object, products, and exponentials.

The category DCPO consisting of dcpos and Scott continuous maps is a Cartesian closed category. However, the category Dom consisting of domains and Scott continuous maps is not. The maximal Cartesian closed categories of domains were characterized by Jung [25]. We are interested in three particular categories: BCD, RB, and FS. Here are descriptions of the Cartesian closed categories of domains we will need.

Definition 1.51. A domain is *bounded complete* if every subset with an upper bound has a least upper bound. Equivalently, a domain is *bounded complete* if every nonempty subset has a greatest lower bound. BCD denotes the category of bounded complete domains and Scott continuous maps.

Definition 1.52. A self-map on a domain D is a *deflation* if it is less than the identity map in the pointwise order and has a finite image.

Definition 1.53. A domain is a *retract of a bifinite domain*, or an *RB-domain*, if there exists a directed family $(f_i)_{i\in I}$ of Scott continuous deflations whose supremum is the identity map. RB denotes the category of RB-domains and Scott continuous maps.

RB deserves its name, because each RB domain is a retract of a domain that can be expressed as a (bi)limit of finite posets.

Definition 1.54. A self-map on a domain D is *finitely separated* from the identity map if there exists a finite set $M \subseteq D$ such that $\forall x \in D, \exists m \in M. f(x) \leq m \leq x$.

Definition 1.55. A domain is a *finitely separated domain*, or an *FS-domain*, if there exists a directed family $(f_i)_{i \in I}$ of Scott continuous self-maps, each finitely separated from the identity map, whose supremum is the identity map. FS denotes the category of FS-domains and Scott continuous maps.

BCD is a subcategory of RB, which is a subcategory of FS. FS is a maximal Cartesian closed category of domains; however, it is a frustrating open question whether RB is a proper subcategory of FS.

Finally, all three of these categories are subcategories of COH, the category of coherent domains and Scott continuous maps, but COH is not Cartesian closed.

1.2.4 Monads

The notion of a monad in category theory gained prominence in the theory of programming languages when Moggi used them to give a denotational semantics for computation effects [1]. Wadler then showed how monads could be implemented in functional programming languages [26]. A monad on a category $\mathcal{C}$ can be thought of as monoid in the category of endofunctors on $\mathcal{C}$.

Definition 1.56. A *monad* on a category $\mathcal{C}$ is a triple, (T, η, μ), where T is an endofunctor and $\eta : \mathrm{Id}_C \to T$, $\mu : T^2 \to T$ are natural transformations such that the following diagrams commute:

The natural transformation η is called the unit of the monad, and μ is the multiplication.

There is an alternate characterization of a monad that uses a Kleisli extension in place of the multiplication. An endofunctor T is a monad if, for any map $f : X \to TY$, there is a Kleisli extension $f^\dagger : TX \to TY$, and the following laws hold:

1. $\eta^\dagger = \mathrm{id}$
2. $h^\dagger \circ \eta_D = h$
3. $k^\dagger \circ h^\dagger = (k^\dagger \circ h)^\dagger$

Given the Kleisli extension, the multiplication is defined by $\mu = \mathrm{id}^{\dagger}_{TX}$. Conversely, given the multiplication and a function $f : X \to TY$, then $f^{\dagger} = \mu \circ T(f)$.

Example 1.57. Consider an endofunctor, M, in the category Set that maps a set A to the set of all words that can be formed using the elements of A as an alphabet. For a function $f : A \to B$, $M(f) : MA \to MB$ is defined so that

$$M(f)(a_1 \dots a_n) = f(a_1) \dots f(a_n)$$

This construction forms the free monoid over a set. A monoid is a set with an associative binary operation and an identity element. In this case, the binary operation is concatenation, and the identity element is the empty word.

This endofunctor is also a monad. The unit $\eta_A : A \to MA$ maps an element $a \in A$ to the word consisting of just the letter a. The multiplication $\mu_A : M^2A \to MA$ takes a word over the alphabet MA, which is a word of words, and forms a single word by concatenating all of the individual words.

1.2.5 Powerdomains

Nondeterminism is modeled in domain theory by powerdomains which are built by considering nondeterministic choice as an idempotent, commutative, and associative operation [27]. This is equivalent to the algebraic definition of a semilattice, so the powerdomains are simply free ordered semilattices over a poset. Powerdomains are so named because they are analogs in domains of the powerset of a set. For example, the powerset of a finite set is the free semilattice monoid over the set.

Suppose P is a poset with an operation $+$ that is commutative and idempotent. If we add the rule that $x \leq x + y$, then P is a sup-semilattice. If $x \geq x + y$, then P is a inf-semilattice, and an ordered semilattice will result from not assuming any relation between x and $x + y$. The lower, or Hoare, powerdomain is the free sup-semilattice over a poset, the upper, or Smyth, powerdomain is the free inf-semilattice over a poset, and the convex, or Plotkin, powerdomain is the free ordered semilattice over a poset. When defined on domains, these powerdomains have nice topological characterizations which will be used in this thesis.

Definition 1.58. For a domain D, the *lower powerdomain* is $\Gamma_0(D)$, the family of nonempty Scott closed subsets of D, ordered by inclusion.

Definition 1.59. For a domain D, the *upper powerdomain* is $SC(D)$, the family of nonempty, saturated, and Scott compact subsets of D, ordered by reverse inclusion.

Definition 1.60. For a domain D, a subset $L \subseteq D$ is a *lens* if L is Scott compact and $L = \overline{L} \cap \uparrow L$, where $\overline{L}$ is the Scott closure of L.

Definition 1.61. For two subsets, A and B of a domain D, we define the *Egli-Milner order* by $A \sqsubseteq_{EM} B \Leftrightarrow A \subseteq \downarrow B \wedge B \subseteq \uparrow A$.

Definition 1.62. For a coherent domain D, the *convex powerdomain* is $Lens(D)$, the family of nonempty lenses, with the Egli-Milner order.

The lower and upper powerdomains form a monad in the categories BCD, RB, and FS, while the convex powerdomain only forms a monad in RB and FS.

1.2.6 Probabilistic Powerdomain

The probability measures on a domain also form a domain. The standard approach to defining the order on measures is via valuations, which we now describe:

Definition 1.63. Let X be a dcpo. A *continuous valuation* on X is a function $\mu : \mathcal{O}(X) \to [0, 1]$ (where $\mathcal{O}(X)$ denotes the Scott open subsets of X) with the following properties:

1. $\mu(\emptyset) = 0$
2. $\mu(O) + \mu(U) = \mu(O \cup U) + \mu(O \cap U)$
3. $O \subseteq U \Rightarrow \mu(O) \leq \mu(U)$
4. $\mu(\bigcup_{i \in I} U_i) = \sup_{i \in I} \mu(U_i)$ for every directed family $(U_i)_{i \in I}$ of open sets.

Definition 1.64. For a dcpo X, the set of all continuous valuations on X, $\mathcal{V}(X)$, is called the *probabilistic powerdomain* of X.

For two continuous valuations μ and μ', the order of $\mathcal{V}(X)$ is defined by $\mu \sqsubseteq \mu'$ if $\mu(O) \leq \mu'(O)$ for all Scott open subsets of X. With this order, $\mathcal{V}(X)$ is a dcpo if X is. For a Scott continuous function $f : X \to Y$, there is a Scott continuous function $\mathcal{V}(f) : \mathcal{V}(X) \to \mathcal{V}(Y)$ defined by:

$$\mathcal{V}(f)(\mu)(O) = \mu(f^{-1}(O))$$

This morphism map allows $\mathcal{V}$ to be a functor in the category DCPO.

Definition 1.65. Let $x \in X$ for some dcpo X. The *point valuation* centered at x, η_x is defined by:

$$\eta_x(O) = \begin{cases} 1 & \text{if } x \in O \\ 0 & \text{otherwise} \end{cases}$$

Definition 1.66. A *simple valuation* on X is a convex combination of point valuations. It can be written as $\sum_{i \in I} r_i \eta_i$, where I is a finite subset of X.

The following lemma from Claire Jones, the Splitting Lemma, is very important in understanding the probabilistic powerdomain of domains.

Lemma 1.67. (Splitting Lemma) *Let μ and ν be simple valuations, so that $\mu = \sum_{i \in I} r_i \eta_i$ and $\nu = \sum_{j \in J} s_j \eta_j$. Then $\mu \sqsubseteq \nu$ if and only if there exist nonnegative numbers $(t_{i,j})_{i \in I, j \in J}$ such that:*

1. *$\sum_{j \in J} t_{i,j} = r_i$ for all i in I.*
2. *$\sum_{i \in I} t_{i,j} \leq s_j$ for all j in J.*
3. *If $t_{i,j} \neq 0$, then $i \sqsubseteq j$ for all i in I and j in J.*

Moreover, $\mu \ll \nu$ if and only if $\leq$ is replaced by $<$ in (2) and $\sqsubseteq$ is replaced by $\ll$ in (3).

Theorem 1.68. *The probabilistic powerdomain of a domain is also a domain.*

The probabilistic powerdomain forms a monad in the category of domains. However, it is not known whether any Cartesian closed category of domains is closed under the probabilistic powerdomain.

1.2.7 Randomized Computation

We consider a randomized computation to be any program or algorithm that uses some source of randomness to guide its computation. This includes assigning a random value to a variable or using a conditional expression that branches based on the output of a random process. If the probability distribution of the random source is known, we can view this as probabilistic computation. Probabilistic computation is usually represented as a probabilistic choice operator, $p +_r q$, where p is chosen with probability r and q is chosen with probability $1 - r$.

Probabilistic Turing machines were first defined by de Leeuw *et al* in 1956 [28]. These machines are the same as normal Turing machines with an attached random device. This device prints 0's and 1's to a tape, with 1's occurring with probability p and 0's occurring with probability

$1 - p$, where $0 < p < 1$. This tape can then be used as an input tape for the Turing machine. It was shown that as long as p is computable, then these machines cannot compute anything that a deterministic machine cannot compute. However, it may be possible that a probabilistic machine can compute something faster than any deterministic machine could [29].

Randomized computation first gained prominence when Rabin [30] introduced a randomized algorithm for finding the nearest pair in a set of n points. This algorithm had a linear average runtime, faster than the $n \log n$ runtime of the fastest known deterministic algorithm. More well known are the algorithms of Solovay and Strassen [31] and Miller and Rabin [32] for determining if a number is prime. These algorithms run in polynomial time (with a small error probability), and they were discovered over 20 years before the AKS primality test [33], the first known deterministic algorithm for recognizing prime numbers in polynomial time.

Chapter 2

A Monad of Random Choice

2.1 Motivation

An important aspect of domain theory is the existence of least fixed points of Scott continuous functions, which are used to model recursion. A typical example is the factorial function:

```
fact(n) = if (n==0) then 1 else n*fact(n-1)
```

Consider the domain $[\mathbb{N}_\perp \to \mathbb{N}_\perp]$, where $\mathbb{N}_\perp$ denotes the flat natural numbers with a bottom element. Functions of this type may be viewed as a sequence of natural numbers (it will be assumed that a function sends the bottom element to itself unless stated otherwise). The identity function can be viewed as $(0, 1, 2, 3, \ldots)$. Now define the Scott continuous function $F : [\mathbb{N}_\perp \to \mathbb{N}_\perp] \to [\mathbb{N}_\perp \to \mathbb{N}_\perp]$ by:

$$F(f)(n) = \begin{cases} \perp & \text{if } n = \perp \\ 1 & \text{if } n = 0 \\ n \cdot f(n-1) & \text{else} \end{cases}$$

where $n \cdot f(n-1) = \perp$ if $f(n-1) = \perp$. The factorial function is simply the least fixed point of F (and the only fixed point). Repeatedly applying F to bottom, which is $(\perp, \perp, \perp, \ldots)$, forms a chain of finite functions, whose supremum is the least fixed point – the factorial function. Though the entire function cannot be evaluated in finite time, when a specific value is needed, it can be obtained by iterating only enough so that the value is defined. For F, $F(\perp, \perp, \ldots) = (1, \perp, \ldots)$. $F^2(\perp, \perp, \ldots) = (1, 1, \perp, \ldots), F^3(\perp, \perp, \ldots) = (1, 1, 2, \perp, \ldots)$, and so on.

Now consider the following randomized function:

```
f(n) = if (n==0) then 0
       else if (coin()==1) then (1 + f(n-1))
       else f(n-1)
```

Here `coin` is just a function that randomly returns either 0 or 1. Thus the function `f` counts the number of heads in n coin flips (where heads $\equiv 1$). This is no longer a deterministic function; each random choice creates a branching of possible outcomes. This function can be represented by an infinite tree of binary words, where each node contains a function of type $[\mathbb{N}_\perp \to \mathbb{N}_\perp]$. The empty word, ϵ, corresponds to not flipping the coin at all and is mapped to $(0, \perp, \perp, \ldots)$, since there can only be zero heads in zero coin flips. For every other n, it returns $\perp$ since n coin flips have not

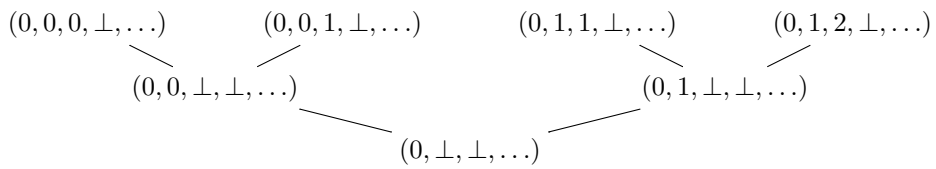


Figure 2.1: Flipping a coin n times

been made yet. The word 0 means that the coin was flipped once and landed tails, so this will be mapped to $(0,0,\bot,\bot,\ldots)$. Similarly, the word 1 means heads and is mapped to $(0,1,\bot,\bot,\ldots)$. The tree can be built up incrementally in this fashion, and these finite trees have a supremum which will represent the entire function. But of course, for any particular n, a concrete representation can be found by using just the $n+1$ lowest levels of the tree. The beginning of this tree is pictured in Figure 2.1.

Similar to the deterministic case, where a recursive function is represented as the supremum of a chain of finite partial functions (given as a fixed point of some functional), the motivation behind the following monad is to provide a setting where a recursive, randomized function can be realized as the supremum of finite binary trees of partial functions.

2.2 The Random Choice Functor

This work is inspired by a model of uniform continuous random variables first proposed by Goubault-Larrecq and Varacca [11]. In their paper, it was shown that the category of bounded complete domains (BCD) is closed under a similar construction. However, their assertion that the construction forms a monad in BCD was incorrect, since the proposed Kleisli extension failed to be monotone, thus not Scott continuous.

The basic idea is to separate the random choices from the domain itself. In the probabilistic powerdomain, the probability distributions are placed on the underlying domain, D. Here, random bits are chosen (0 and 1), forming a binary tree, M, of all possible choices. Then a function, f, is defined from these random choices of bits to the underlying domain (so $f : M \to D$). In the probabilistic powerdomain, for an element d, making a choice between d and d ($d \oplus d$) is the same as just d, since the probabilities are the same. Here, there is distinction between $d \oplus d$ and d. In the first case, a random bit is still chosen, so programmatically, this is distinct from the latter case where no such choice is made.

Let $\{0,1\}^\infty = \{0,1\}^* \cup \{0,1\}^\omega$ be the set of finite and infinite words of alphabet $\{0,1\}$, with

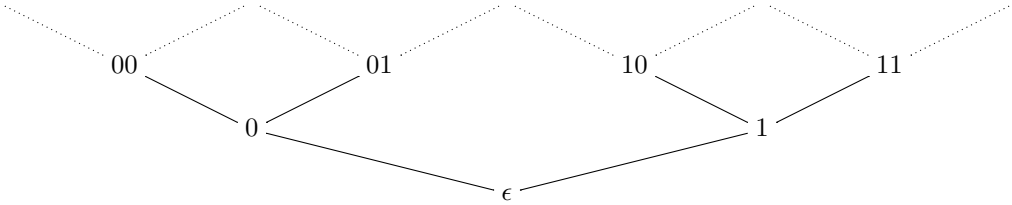


Figure 2.2: Hasse diagram of $\{0,1\}^\infty$

the prefix order ($w \leq w'$ if w is a prefix of w'). The symbol $*$ is used to denote the concatenation operation. In this setting, a 0 represents getting tails on a coin flip, and a 1 signifies heads. The probabilities associated with the coin do not need to be specified here. Any coin or oracle that returns random bits will work. Of course, the probability distribution of the random oracle will determine the probabilistic behavior of the resulting randomized algorithm.

2.2.1 Properties of $\{0,1\}^\infty$

The set of finite and infinite words with alphabet $\{0,1\}$ is clearly a poset with the prefix order and a dcpo since it includes the infinite words. The first lemma is obvious.

Lemma 2.1. *The prefixes of a word form a chain. If two words, w and w', are both prefixes of another word z, then $w \leq w'$ or $w' \leq w$. Therefore, any directed set in $\{0,1\}^\infty$ must actually be a chain.*

Lemma 2.2. *$\{0,1\}^\infty$ is a dcpo.*

Proof. Any directed set is a chain. If the chain only has finitely many distinct words, then the biggest one is the supremum. If there are infinitely many distinct words, they must all be under some infinite word, $w \in \{0,1\}^\omega$, which is the supremum. ■

Here are some more properties of $\{0,1\}^\infty$ that will prove useful.

Lemma 2.3. *The finite elements of $\{0,1\}^\infty$, $K(\{0,1\}^\infty)$, are precisely the finite words.*

Proof. A finite word only has finitely many prefixes. Thus, a directed set, or chain, whose supremum is above a finite word w must contain a word above w. If a word is infinite, then its finite prefixes form a chain whose supremum is the infinite word and yet does not contain that infinite word. ■

Proposition 2.4. *$\{0,1\}^\infty$ is an algebraic domain.*

Proof. Every word, finite or infinite, is the supremum of its finite prefixes. These prefixes form a chain, so they are directed. ■

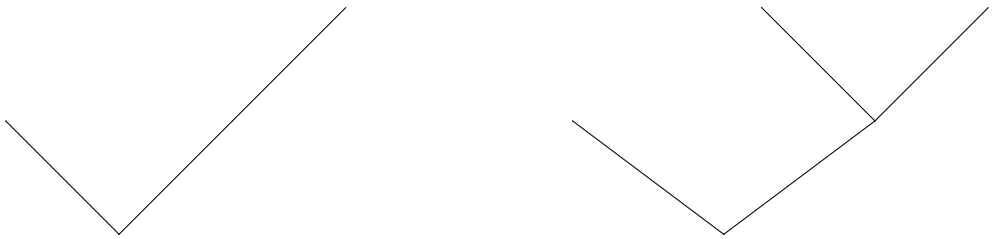


Figure 2.3: The top of the left tree does not form a full antichain, but the right tree does.

The Scott topology of an algebraic domain has a basis consisting of the upper sets of finite elements. Thus, the set $\{\uparrow w \mid w \in \{0,1\}^*\}$ is a basis for the Scott topology of $\{0,1\}^\infty$. Any open set is of the form $\bigcup_{w \in K} \uparrow w$, for some $K \subseteq \{0,1\}^*$.

Proposition 2.5. *$\{0,1\}^\infty$ is Scott compact.*

Proof. Any open covering of $\{0,1\}^\infty$ must contain the empty word. However, the only Scott open set containing the empty word is $\{0,1\}^\infty$ itself. Thus, $\{0,1\}^\infty$ is Scott compact. ■

Proposition 2.6. *$\{0,1\}^\infty$ is coherent.*

Proof. By Proposition 4.38 of [27], a Scott compact domain P is coherent if and only if for any finite sets, $F, G \subseteq K(P)$, there exists another finite set, $H \subseteq K(P)$ such that $\uparrow F \cap \uparrow G = \uparrow H$. If this intersection is empty, then setting H as the empty set satisfies this equality. If F and G are finite sets of finite words, then let $H = \{w \in F \mid \exists z \in G, z \leq w\} \cup \{w \in G \mid \exists z \in F, z \leq w\}$. If $w \in \uparrow F \cap \uparrow G$, then there is a $v \in F$ such that $v \leq w$ and a $z \in G$ such that $z \leq w$. Since v and z are both prefixes of w, they must compare. If $v \leq z$, then by the definition of H, $z \in H$. Similarly, if $z \leq v$, then $v \in H$. Therefore, either v or z is in H, so $w \in \uparrow H$. Conversely, if $w \in \uparrow H$, then $z \leq w$ for some $z \in H$. Again, by the definition of H, z is either in F and above some word in G or it is in G and above some word in F. In either case, w is above some word of both F and G, so $w \in \uparrow F \cap \uparrow G$. ■

2.2.2 Definition of the Functor

Definition 2.7. An *antichain* of $\{0,1\}^\infty$ is a subset of words such that no two distinct words are comparable (no word is a prefix of another word). A nonempty antichain M is considered *full* if $\forall w \in \{0,1\}^\omega, \exists z \in M, z \leq w$. Alternatively, $\{0,1\}^\omega \subseteq \uparrow M$, or $M \sqsubseteq_{EM} \{0,1\}^\omega$. Denote the full antichains by $FAC(\{0,1\}^\infty)$.

Using coin flips in a program results in a branching of computation that can be represented as a binary tree. The final possible outcomes will be located at the leaves of this tree, which must form an antichain. This antichain is required to be full since for any coin flip, it is assumed that either heads or tails appears, and both outcomes must be accounted for. The idea of using antichains comes from [11].

Definition 2.8. For a category of domains, the functor of random choice, RC, is defined on objects by

$$RC(D) = \{(M, f) \mid M \in FAC(\{0,1\}^\infty), f : M \to D\}$$

where f is Scott continuous, giving M the subspace topology from the Scott topology of $\{0,1\}^\infty$.

For a morphism $a : D \to D'$ and $(M, f) \in RC(D)$, the functor acts on a as follows:

$$RC(a)(M, f) = (M, a \circ f)$$

For a domain D, we order $RC(D)$ by $(M, f) \sqsubseteq (N, g)$ if and only if $M \sqsubseteq_{EM} N$ and $w \leq z \Rightarrow f(w) \sqsubseteq g(z), \forall w \in M, z \in N$. Since the antichains are required to be full, $M \sqsubseteq_{EM} N$ is equivalent to $M \subseteq {\downarrow}N$. If M is an antichain and $M \sqsubseteq_{EM} N$, then for each $z \in N$, there is exactly one $w \in M$ such that $w \leq z$. Thus, we can project the words of N down to the words of M in a unique, well defined manner. We call this function π_M. Using this, another characterization for the order on functions is $\forall z \in N, f \circ \pi_M(z) \sqsubseteq g(z)$.

For (M, f) in $RC(D)$, we will use π_1 and π_2 to single out the individual components. $\pi_1(M, f) = M$ and $\pi_2(M, f) = f$.

Now it will be shown that if D is a bounded complete domain, then so is $RC(D)$. A proof of this for a similar structure can be found in [12]. In fact, many of the following proofs fall easily from results found there. First, we consider the full antichains, $FAC(\{0,1\}^\infty)$.

2.2.3 Properties of $FAC(\{0,1\}^\infty)$

Lemma 2.9. *For any antichain $X \in FAC(\{0,1\}^\infty)$, ${\downarrow}X$ is Scott closed. Furthermore, X is Lawson closed.*

Proof. ${\downarrow}X$ is obviously a lower set, so we just need to check the suprema of directed sets. In $\{0,1\}^\infty$, all directed sets are simply chains (Lemma 2.1). The only way that one of these chains, D, can fail to contain its supremum is if the supremum of D is an infinite word, $w \in \{0,1\}^\omega$. Since X

is a full antichain, it contains exactly one z such that $z \leq w$. If z were finite, then $\downarrow X$ could not contain D since D must contain words above z (and under w). Thus, z must be w, and X is Scott closed. By Proposition 3.4 in [34], X is Lawson closed. ■

Lemma 2.10. *$FAC(\{0,1\}^\infty)$ is a subset of the convex powerdomain, $\mathcal{P}_C(\{0,1\}^\infty)$.*

Proof. For $X \in FAC(\{0,1\}^\infty)$, it must be the case that $X = \overline{X} \cap \uparrow X = \downarrow X \cap \uparrow X$ (since $\downarrow X$ is closed). For any $d \in \downarrow X \cap \uparrow X, \exists x \in X$ so that $d \leq x$ and $\exists y \in X$ so that $y \leq d$. Since X is an antichain, $y \leq d \leq x$ implies that $y = d = x$, so $\downarrow X \cap \uparrow X \subseteq X$. The other direction of containment is obvious. ■

The following proposition is well known for the convex powerdomain [27].

Proposition 2.11. *If D is a coherent domain, then so is $\mathcal{P}_C(D)$.*

Corollary 2.12. *$\mathcal{P}_C(\{0,1\}^\infty)$ is a coherent domain.*

Now we show that $FAC(\{0,1\}^\infty)$ is a sub-dcpo of $\mathcal{P}_C(\{0,1\}^\infty)$.

Lemma 2.13. *Suppose that $X \in FAC(\{0,1\}^\infty)$ and that $S \in \mathcal{P}_C(\{0,1\}^\infty)$. If $X \sqsubseteq_{EM} S$, then $X \sqsubseteq_{EM} \mathrm{Min}(S)$.*

Proof. It is given that $S \subseteq \uparrow X$ and $X \subseteq \downarrow S$. Since $S \subseteq \uparrow X$, $\mathrm{Min}(S) \subseteq \uparrow X$. What is left to show is that $X \subseteq \downarrow \mathrm{Min}(S)$. Suppose there is some $x \in X$ not below any word of $\mathrm{Min}(S)$. $X \subseteq \downarrow S$, so $\exists s \in S$ such that $x \leq s$. There is also some word $m \in \mathrm{Min}(S)$ such that $m \leq s$. Both x and m are prefixes of s, so they must compare, and since $x \notin \downarrow \mathrm{Min}(S)$, $m < x$. However, $\mathrm{Min}(S) \subseteq \uparrow X$, so $\exists y \in X$ such that $y \leq m$. But this implies $y < x$ which contradicts X being an antichain. Therefore, the claim holds. ■

Lemma 2.14. *$FAC(\{0,1\}^\infty)$ is a dcpo.*

Proof. Since $\{0,1\}^\infty$ is a dcpo, the convex powerdomain, $\mathcal{P}_C(\{0,1\}^\infty)$, is a dcpo, and $FAC(\{0,1\}^\infty)$ is a subset that uses the same order. Let D be a directed set in $FAC(\{0,1\}^\infty)$. There must a supremum, S, in $\mathcal{P}_C(\{0,1\}^\infty)$. All that is left to show is that S is a full antichain. For every element $d \in D$, $d \sqsubseteq_{EM} \{0,1\}^\omega$. Therefore, $S \sqsubseteq_{EM} \{0,1\}^\omega$, and S is full. If S were not an antichain, it could be replaced with $\mathrm{Min}(S)$, the minimal elements of S. $\mathrm{Min}(S) \subseteq \downarrow S$ and $S \subseteq \uparrow \mathrm{Min}(S)$, so $\mathrm{Min}(S) \sqsubseteq_{EM} S$. From the previous lemma, if S is an upper bound for D, then so is $\mathrm{Min}(S)$. Thus, S could not be the supremum. ■

In fact, $FAC(\{0,1\}^\infty)$ is a complete lattice, as shown here.

Lemma 2.15. *For any two antichains, $X, Y \in FAC(\{0,1\}^\infty)$, the supremum of X and Y, $X \vee Y$ is equal to* $\mathrm{Max}(X \cup Y)$.

Proof. Clearly, $X, Y \subseteq \downarrow \mathrm{Max}(X \cup Y)$. Now to show $\mathrm{Max}(X \cup Y) \subseteq \uparrow X \cap \uparrow Y$, let m be in $\mathrm{Max}(X \cup Y)$. Without loss of generality, let $m \in X$ (so $m \in \uparrow X$). It must be shown that $m \in \uparrow Y$. Pick any $z \in \{0,1\}^\omega$ such that $m \leq z$. Since Y is full, there is some $y \in Y$ such that $y \leq z$. Thus, y and m are comparable. Since m is maximal, it cannot be the case that $m < y$. Therefore, $y \leq m$, and $m \in \uparrow Y$.

This shows that $X, Y \sqsubseteq_{EM} \mathrm{Max}(X \cup Y)$. Now let $X, Y \sqsubseteq_{EM} Z$. Then $X \cup Y \subseteq \downarrow Z$, so $\mathrm{Max}(X \cup Y) \subseteq \downarrow Z$. Therefore, $\mathrm{Max}(X \cup Y) \sqsubseteq_{EM} Z$. ■

Proposition 2.16. *$FAC(\{0,1\}^\infty)$ is a complete lattice.*

Proof. The above lemma shows that any nonempty subset in $FAC(\{0,1\}^\infty)$ can be made directed by adding the supremum of each pair of elements without changing the supremum of the set. Since $FAC(\{0,1\}^\infty)$ is a dcpo, every nonempty subset has a supremum. The antichain, $\{\epsilon\}$, only containing the empty word, is the bottom element, which is the supremum of the empty set. Thus, all subsets have a supremum, and $FAC(\{0,1\}^\infty)$ is a complete lattice. ■

Fact 2.17. *For any nonempty subset $\{M_i\}$ in $FAC(\{0,1\}^\infty)$, the supremum of the subset can be found by $\bigsqcup_i M_i = \mathrm{Min} \bigcap_i \uparrow M_i$.*

The following two lemmas characterize the finite elements of $FAC(\{0,1\}^\infty)$.

Lemma 2.18. *If an antichain $X \in FAC(\{0,1\}^\infty)$ only contains finite words, then it is a finite set.*

Proof. X is a Lawson closed subset of $\{0,1\}^\infty$ by Lemma 2.9, which is coherent (thus Lawson compact). Therefore, X is Lawson compact. The set $\{\uparrow w \mid w \in X\}$ forms an open cover of X since each w is finite. There must be a finite subcover, and since X is an antichain, X must be finite. ■

The following theorem from [34] will be useful in showing that $FAC(\{0,1\}^\infty)$ is a domain.

Theorem 2.19. *Let A be a finite set, and for each n, let $\pi_n : A^\infty \to A^{\leq n} \equiv \{s \in A^* \mid |s| \leq n\}$ be the projection onto the set of words of length at most n. Then π_n is continuous for each n, where we endow A^∞ and $A^{\leq n}$ with either the Scott or Lawson topologies. Moreover,*

1. *Each Lawson-compact antichain $X \subseteq A^\infty$ satisfies $\{\pi_n(X)\}_n$ is a directed family of finite antichains satisfying* $\sup_n \pi_n(X) = X$.

2. *Conversely, each directed family of finite antichains* $F_n \subseteq A^{\leq n}$ *satisfies* $\sup_n F_n = X$ *is a Lawson-compact antichain in* A^∞ *satisfying* $\pi_n(X) = F_n$ *for each* n.

Lemma 2.20. *The finite elements of* $FAC(\{0,1\}^\infty)$ *are precisely the antichains containing only finite words.*

Proof. A full antichain with only finite words has finitely many elements by Lemma 2.18, so there is a finite number of antichains below it. Therefore, it must be finite. If an antichain X contains an infinite word, then $\{\pi_n(X)\}_n$ from the above theorem is a directed family of full antichains whose supremum is X. Everything in the directed family is finite, so it does not contain X. Therefore, X is not finite. ■

Proposition 2.21. $FAC(\{0,1\}^\infty)$ *is a algebraic domain.*

Proof. It must be shown that for any $X \in FAC(\{0,1\}^\infty)$, $X = \bigsqcup\big(\downarrow X \cap K(FAC(\{0,1\}^\infty))\big)$. The directedness is trivial since the supremum of two finite antichains is also finite, by Lemma 2.15. Clearly, $\bigsqcup\big(\downarrow X \cap K(FAC(\{0,1\}^\infty))\big) \sqsubseteq_{EM} X$. For the other direction, again consider the family $\{\pi_n(X)\}_n$, whose elements are all finite and below X. Then

$$\bigsqcup\big(\downarrow X \cap K(FAC(\{0,1\}^\infty))\big) \sqsupseteq_{EM} \bigsqcup\{\pi_n(X)\}_n = X$$

Thus, $FAC(\{0,1\}^\infty)$ is algebraic. ■

2.2.4 Properties of the RC Functor

Now that properties of the antichains have been proven, attention must turn to the second component of the RC functor: the functions. Again, for an element $(M, f) \in RC(D)$, the function f must be continuous from the relative Scott topology of M to the Scott topology of D. This continuity restricts where a function can send an infinite word. If a function has a sudden jump upwards at an infinite word, it will fail to be continuous. An example of this can be seen in Figure 2.4.

A certain characterization of continuous functions will be useful in proving some pairs, (M, f), are actually elements of $RC(D)$ for some D. A function $f : M \to D$ is continuous at a point w if for any open set, U, of D containing $f(w)$, there is an open set, V, of M containing w such that $f(V) \subseteq U$. A function is continuous if and only if it is continuous at every point. Note that for

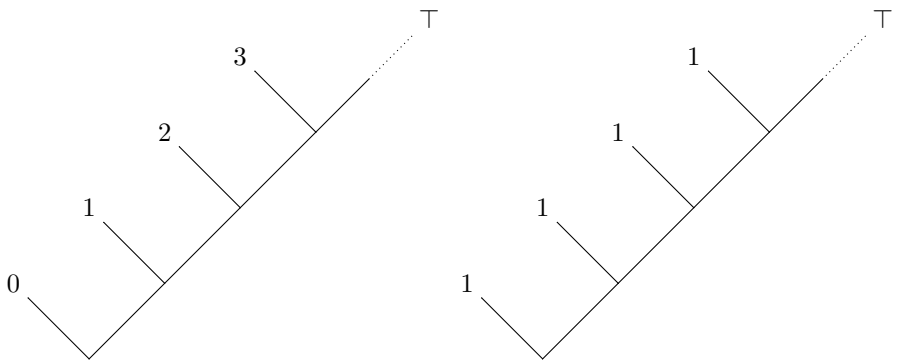


Figure 2.4: The function on the left antichain is Scott continuous, but the function on the right is not.

a function $f : FAC(\{0,1\}^\infty) \to D$, f is continuous at any finite word, w, since $\uparrow w$ is an open set containing w.

Lemma 2.22. *If two elements of $RC(D)$, (M, f) and (N, g), have an upper bound, then for all $w \in M, v \in N$, if w and v are comparable, then $f(w)$ and $g(v)$ have an upper bound in D.*

Proof. If there are comparable words, w and v, such that $f(w)$ and $g(w)$ have no upper bound, then for any z above w and v, there is no value for $h(z)$ that can make (L, h) an upper bound. ■

Corollary 2.23. *If $\{M_i, f_i\}$ is a directed family in $RC(D)$, then for any chain of words $\{w_i\}$, where $w_i \in M_i$, the family $\{f_i(w_i)\}$ is directed.*

The previous lemma and corollary can now be used to show that RC is an endofunctor on the category of dcpos.

Proposition 2.24. *If D is a dcpo, then so is $RC(D)$.*

Proof. Let $\{(M_i, f_i)\}$ be a directed family in $RC(D)$. From the previous section, the family $\{M_i\}$ has a supremum, M. Now for any $w \in M$, each M_i contains a w_i below w. Each pair of these, w_i and w_j, are comparable, and since $\{(M_i, f_i)\}$ is directed, then $f_i(w_i)$ and $f_j(w_j)$ have an upper bound in D. Thus, the family $\{f_i(w_i)\}$ is directed by Corollary 2.23, and because D is a dcpo, it has a supremum, d_w. Therefore, (M, f), where $f(w) = d_w$ is the supremum of the directed family $\{(M_i, f_i)\}$, as long as f is continuous in the relative Scott topology of M. Suppose $d \ll f(w) = \bigsqcup_i f_i \circ \pi_{M_i}(w)$. Since $\Uparrow d$ is open, there is an i such that $d \ll f_i \circ \pi_{M_i}(w)$. Since f_i is continuous, $f_i^{-1}(\Uparrow d)$ is open of the form $(\bigcup_j w_j) \cap M_i$ and $w_j \leq w$ for some j. Because $(M_i, f_i) \sqsubseteq (M, f)$, $f(\uparrow w_j \bigcap M) \subseteq \Uparrow d$, so f is continuous. ■

A function continuous in the relative Scott topology can be decreased at a finite number of words and still remain continuous.

Lemma 2.25. *Suppose (M, f) is in $RC(D)$, where D is a domain. Define $f' : M \to D$ as follows:*

$$f'(z) = \begin{cases} d & \text{if } z = w \\ f(z) & \text{if } z \neq w \end{cases}$$

where w is some word in M and d is some element of D such that $d \sqsubseteq f(w)$. Then, $(M, f') \in RC(D)$ (f' is still continuous with the subspace topology on M).

Proof. Since M is an antichain, all words are separated by open sets. For any $z \neq w$, there is an open set containing z but not w. Therefore, altering the function f at word w will not change the continuity at z. All that needs to be shown is that f' is still continuous at w. Suppose $f'(w) \subseteq \uparrow d$ for some $d \in D$. Then $f(w) \in \uparrow d$. Since f is continuous, there is an open set, U, containing w such that $f(U) \subseteq \uparrow d$. Since open sets are of the form $\bigcup(\uparrow z \cap M)$, there is a z such that $w \in \uparrow z$ and $f(\uparrow z \cap M) \subseteq \uparrow d$. Since $f'(w) \in \uparrow d$ and for all other words, f' is equal to f, then $f'(\uparrow z \cap M) \subseteq \uparrow d$. Thus f' is continuous. ■

A necessary requirement can be stated for the way below relation for $RC(D)$.

Proposition 2.26. *Let D be a bounded complete domain. If $(M, f) \ll (N, g)$ in $RC(D)$, then for $w \in M$, $z \in N$ such that $w \leq z$, $f(w) \ll g(z)$.*

Proof. Suppose there exists a $w \in M$ such that $w \leq z \in N$ and suppose that $f(w)$ is not way below $g(z)$. Then there exists a directed family $\{d_i\}$ so that $g(z) \sqsubseteq \bigsqcup\{d_i\}$, but there is no d_i such that $f(w) \sqsubseteq d_i$. Make a new directed family $\{e_i\}$, where $e_i = g(z) \wedge d_i$. Then $\bigsqcup\{e_i\} = g(z)$, but no e_i is above $f(w)$. Now create a directed family $\{(N, g_i)\}$, where (N, g_i) is identical to (N, g) except for the one particular z, $g_i(z) = e_i$. From the above lemma, these functions are still continuous since $e_i \sqsubseteq g(z)$. This directed family has a supremum, (N, g), but no (N, g_i) is above (M, f). This contradicts that $(M, f) \ll (N, g)$, so the claim must hold. ■

The following proposition will be used to demonstrate an analogue to Theorem 2.19.

Proposition 2.27. *Suppose D is a bounded complete domain and $(M, f) \in RC(D)$. Then the function $\overline{f} : \downarrow M \to D$, defined by $\overline{f}(w) = \inf f(\uparrow w \cap M)$ is Scott continuous.*

Proof. Clearly, $\overline{f}$ is monotone. Fix a w in $\downarrow M$ and suppose $\overline{f}(w) \in \uparrow d$ for some $d \in D$. If w is finite, then $\uparrow w$ is an open set containing w such that $\overline{f}(\uparrow w) \subseteq \uparrow d$. Now assume w is an infinite word. Then $\overline{f}(w) = f(w)$. By the Interpolation Lemma (Theorem 1.22) for domains, there exists some d' such that $d \ll d' \ll f(w)$. Since f is continuous, there is an open set, U, containing w such that $f(U) \subseteq \uparrow d'$. The open set must be of the form $\bigcup \uparrow z \cap M$, so there is a z so that $w \in \uparrow z$ and $f(\uparrow z \cap M) \subseteq \uparrow d'$. Then, $\overline{f}(z) = \inf f(\uparrow z \cap M) \sqsupseteq d'$ since it is the infimum of elements all above d'. Thus, $\overline{f}(\uparrow z) \subseteq \uparrow d' \subseteq \uparrow d$. Since sets of the form $\uparrow d$ are a basis for the topology of D, $\overline{f}$ is continuous at w, and since w was arbitrary, $\overline{f}$ is continuous. ■

Corollary 2.28. *For a bounded complete domain D and $(M, f) \in RC(D)$, define $\pi_n(M, f) = (\pi_n(M), \pi_n(f))$, where $\pi_n(M)$ is the projection of M onto the set of words of length at most n, and $\pi_n(f)(w) = \inf f(\uparrow w \cap M)$. Then $(M, f) = \bigsqcup_n \pi_n(M, f)$.*

Corollary 2.29. *Suppose D is a bounded complete domain and $(M, f), (N, g)$ are two elements of $RC(D)$. If $(M, f) \ll (N, g)$, then M is finite.*

Proof. From the above corollary, $(N, g) = \bigsqcup_n \pi_n(N, g)$, so it is the directed supremum of elements all with finite antichains. If $(M, f) \ll (N, g)$, then (M, f) is below some $\pi_n(N, g)$. Thus, M must also be finite.

With this, a sufficient condition for the way below relation of $RC(D)$ can be stated.

Proposition 2.30. *Let D be a bounded complete domain and $(M, f), (N, g) \in RC(D)$. If M is finite with $M \sqsubseteq_{EM} N$, and if for any $w \in M$, $f(w) \ll \inf g(\uparrow w \cap N)$, then $(M, f) \ll (N, g)$.*

Proof. Let $\{(L_i, h_i)\}$ be a directed family with $(N, g) \sqsubseteq \bigsqcup_i (L_i, h_i)$. Then $\{(L_i)\}$ is a directed family of antichains, with $N \sqsubseteq_{EM} \bigsqcup_i (L_i)$, so there exists some j such that $M \sqsubseteq_{EM} L_j$. Now only consider elements of $\{(L_i, h_i)\}$ such that $M \sqsubseteq L_i$. For any w in M, $\{\inf h_i(\uparrow w \cap L_i)\}$ is a directed family and $\bigsqcup_i \inf h_i(\uparrow w \cap L_i) = \inf \bigsqcup_i h_i(\uparrow w \cap L_i) \sqsupseteq \inf g(\uparrow w \cap N)$. Thus, there is some (L_w, h_w) such that $f(w) \sqsubseteq \inf g(\uparrow w \cap N)$ This can be done for each w in M, and M is finite. The family $\{(L_i, h_i)\}$ is directed, so it contains an element (L_z, h_z) that is above all such (L_w, h_w). Therefore, $(M, f) \sqsubseteq (L_z, h_z)$, and $(M, f) \ll (N, g)$. ■

Now enough is known about the way below relation to show that the random choice functor applied to a domain results in another domain.

Proposition 2.31. *If D is a bounded complete domain, then $RC(D)$ is a domain.*

Proof. Given $(M, f) \in RC(D)$, let (N_1, g_1) and (N_2, g_2) be two elements way below (M, f). To show that $\downarrow(M, f)$ is directed, an upper bound must exist for (N_1, g_1) and (N_2, g_2) that is also way below (M, f). From Lemma 2.15, there is a least upper bound, L, of N_1 and N_2, which must be below M. For each $z \in L$, there is a v_1 in N_1 and a v_2 in N_2 below z. Any word, w, in M above z is also above v_1 and v_2. Since both (N_1, g_1) and (N_2, g_2) are way below (M, f), $g_1(v_1) \ll f(w)$ and $g_2(v_2) \ll f(w)$ by Proposition 2.26. Thus, $f(w)$ is a upper bound for $g_1(v_1)$ and $g_2(v_2)$, and since D is a bounded complete domain, there is a least upper bound for $g_1(v_1)$ and $g_2(v_2)$. Now define (L, h) so that $h(z)$ is the least upper bound of $g_1(v_1)$ and $g_2(v_2)$. Since L is finite, h is continuous in the relative Scott topology.

Clearly, (L, h) is above both (N_1, g_1) and (N_2, g_2). To prove that $\downarrow(M, f)$ is directed, it suffices to show that (L, h) is also way below (M, f). Let $\{(M_i, f_i)\}$ be a directed family whose supremum is above (M, f). Then there must be some (M_j, f_j) above (N_1, g_1) and some (M_k, f_k) above (N_2, g_2). $\{(M_i, f_i)\}$ is directed, so there is a (M_p, f_p) above both (N_1, g_1) and (N_2, g_2). This means $N_1 \sqsubseteq M_p$ and $N_2 \sqsubseteq M_p$, and because L is a least upper bound, $L \sqsubseteq M_p$. For any $z \in L$ and $w \in M_p$ such that $z \leq w$, again let $v_1 \in N_1$ and $v_2 \in N_2$ be both below z. Then $g_1(v_1) \sqsubseteq f_p(w)$ and $g_2(v_2) \sqsubseteq f_p(w)$, and since $h(z)$ is the least upper bound, $h(z) \sqsubseteq f_p(w)$. Therefore, $(L, h) \sqsubseteq (M_p, f_p)$ and $\downarrow(M, f)$ is directed.

Finally, we show that $(M, f) = \bigsqcup \downarrow(M, f)$. Consider the projections to finite antichains, as described in Corollary 2.28. Let $(M_n, f_n) = \pi_n(M, f)$. For a fixed n, consider a subset of $RC(D)$, $\{(M_n, g) \mid g(w) \ll f_n(w), \forall w \in M_n\}$. Each element of the set is way below (M_n, f_n) and since D is a domain, $(M_n, f_n) = \bigsqcup\{(M_n, g) \mid g(w) \ll f_n(w), \forall w \in M_n\}$. Finally, since $(M, f) = \bigsqcup_n (M_n, f_n)$, then $(M, f) = \bigsqcup_n \bigsqcup\{(M_n, g) \mid g(w) \ll f_n(w), \forall w \in M_n\}$. Thus, (M, f) is the directed supremum of some elements way below itself, so it must be true that $(M, f) = \bigsqcup \downarrow(M, f)$. ■

Finally, it can be shown that RC is an endofunctor in the category BCD.

Theorem 2.32. *If D is a bounded complete domain, then so is $RC(D)$.*

Proof. Let $\{(M_i, f_i)\}$ be a subset of $RC(D)$ bounded by some (N, g). Then $M_i \sqsubseteq_{EM} N$ and $f_i \circ \pi_{M_i}(w) \sqsubseteq g(w)$ for each w in N, for all i. The set $\{M_i\}$ must have a supremum, M, such that $M \sqsubseteq_{EM} N$. Any w in M must be below some z in N. Then the set $\{f_i \circ \pi_{M_i}(w)\}$ is bounded by $g(z)$, and since D is a bounded complete domain, its supremum exists. Thus, the candidate for the supremum of $\{(M_i, f_i)\}$ is (M, f), where $f(w) = \bigsqcup_i f_i \circ \pi_{M_i}(w)$.

All that is left to check is that f as defined is continuous in the relative Scott topology of

M. Consider a basic open set in D, $\uparrow d$. For each f_i, $f_i^{-1}(\uparrow d)$ is open, of the form $(\bigcup_{w_i} \uparrow w_i) \cap M_i$. Then $\bigcup_i (\bigcup_{w_i} \uparrow w_i) \cap M$ is open in the relative Scott topology of M. If this set is equal to $f^{-1}(\uparrow d)$, then f is continuous. Suppose $z \in \uparrow w_i \cap M$ for some i, so that $d \ll f_i \circ \pi_{M_i}(z)$. Then because $f(z)$ is the supremum of all such elements, $d \ll f(z)$. If z is in M but not above any such w_i, then $f_i \circ \pi_{M_i}(z)$ is not way above d for any i. The set of such elements, $\{f_i \circ \pi_{M_i}(z)\}$, is bounded by $f(z)$, so it can be made directed by adding in the suprema of all pairs of elements. As the supremum of these elements, $f(z)$ cannot be way above d. Therefore, $f^{-1}(\uparrow d) = \bigcup_i (\bigcup_{w_i} \uparrow w_i) \cap M$, and f is continuous. ■

2.3 The RC Monad

To show that the functor RC forms a monad, the unit and Kleisli extension (or multiplication) of the monad must be exhibited. For a domain D and $d \in D$, the unit, $\eta : D \to RC(D)$ is defined by

$$\eta(d) = (\epsilon, \chi_d)$$

where ϵ is the antichain only containing the empty word, and χ_d is the constant function whose value is d.

2.3.1 First attempt at a Kleisli extension

For any function $h : D \to RC(E)$, the Kleisli extension lifts the function to another function $h^\dagger : RC(D) \to RC(E)$. An element of $RC(D)$ is of the form (M, f) with $f : M \to D$. Composing h with f gives $(M, h \circ f)$, which is an element of $RC(RC(E))$. In essence, this is a random choice of random choices. The duty of the Kleisli extension is to flatten this into one random choice, an element of $RC(E)$.

The Kleisli extension used by Goubault-Larrecq and Varacca for their uniform continuous random variables is defined as follows:

$$h^\dagger(M, f) = (M^\dagger, f^\dagger)$$

where

$$M^\dagger = \bigcup_{w \in M} \bigcup_{w' \in \pi_1 \circ h \circ f(w)} w * w'$$

$$f^\dagger(w * w') = (\pi_2 \circ h \circ f(w))(w')$$

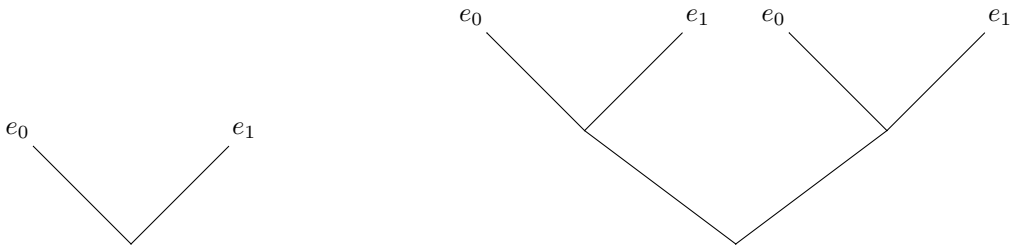


Figure 2.5: These two trees are the results of the first Kleisli extension being applied to two comparable trees. However, these trees are not comparable, so the Kleisli extension is not monotone.

The first component, $M^\dagger$, is an antichain in $FAC(\{0,1\}^\infty)$. It is composed of all words w in the antichain M concatenated with each word w' of the antichain $\pi_1 \circ h \circ f(w)$. The second component is a function $f^\dagger : M^\dagger \to E$. Each word of $M^\dagger$ is of the form $w * w'$. Since $w \in M$ and $h \circ f$ is a function from M to $RC(E)$, the function $\pi_2 \circ h \circ f(w)$ sends each w' in $\pi_1 \circ h \circ f(w)$ to E.

This Kleisli extension satisfies the monad laws at the object level [34], but it has one major problem: it is not monotone. Since we are working in categories of domains with Scott continuous maps, the Kleisli extension must be Scott continuous. Here is a counterexample showing that $h^\dagger$ is not monotone:

For domains D and E, let $h : D \to RC(E)$ be the constant function that sends everything to $(\{0,1\}, g)$, where $g(0) = e_0$, $g(1) = e_1$, and e_0 and e_1 are not comparable. Now consider two elements of $RC(D)$, (ϵ, f_1) and $(\{0,1\}, f_2)$, where $f_1(\epsilon)$ is below both $f_2(0)$ and $f_2(1)$. Therefore, $(\epsilon, f_1) \sqsubseteq (0 \cup 1, f_2)$.

$h^\dagger(\epsilon, f_1) = (\{0,1\}, g)$, while $h^\dagger(\{0,1\}, f_2) = (\{00, 01, 10, 11\}, g')$ where $g'(00) = g'(10) = e_0$ and $g'(01) = g'(11) = e_1$.

The output of the Kleisli extension can be seen in Figure 2.5. If $h^\dagger$ is monotone, then $(\{0,1\}, g) \sqsubseteq (\{00, 01, 10, 11\}, g')$. It is true that $\{0,1\} \sqsubseteq_{EM} \{00, 01, 10, 11\}$. However, since $0 \leq 01$, $g(0) = e_0$ should be less than $g'(01) = e_1$, but they are incomparable. Therefore, $h^\dagger$ is not monotone and this Kleisli extension is not valid in any category that uses Scott continuous functions.

2.3.2 Kleisli Extension of the Monad

The above Kleisli extension is not monotone for the same reason that concatenation of words is not monotone with respect to the prefix order. For each word w, the extension concatenated w with each word of $\pi_1 \circ h \circ f(w)$. In order to define a Kleisli extension that is monotone, concatenation should not be used.

Consider $h : D \to RC(E)$. To define the extension $h^\dagger : RC(D) \to RC(E)$ on (M, f), $h \circ f(w)$ must be considered for each $w \in M$. In the previous, non-monotone, extension, the entire tree of $\pi_1 \circ h \circ f(w)$ factored into the extension. Instead, our new extension only considers the part of $\pi_1 \circ h \circ f(w)$ that is on the same "branch" of the tree as w, namely $\uparrow w \cup \downarrow w$.

Our new candidate for $h^\dagger$ is defined as follows:

$$\pi_1 \circ h^\dagger(M, f) = \bigcup_{w \in M} \mathrm{Min}(\uparrow w \cap \uparrow \pi_1 \circ h \circ f(w))$$

$$((\pi_2 \circ h^\dagger)(M, f))(z) = g(\pi_N(z)) \text{ where } (N, g) = h \circ f \circ \pi_M(z)$$

where $\mathrm{Min}(W)$ denotes the minimal words of W.

Proposition 2.33. *$h^\dagger$ is monotone.*

Proof. $(M, f) \leq (N, g)$ means that $N \subseteq \uparrow M$ (thus, $\uparrow N \subseteq \uparrow M$) and $w \leq z \Rightarrow f(w) \leq g(z)$ for any $w \in M, z \in N$.

$$\begin{aligned}
\uparrow \pi_1 \circ h^\dagger(M, f) &= \uparrow \bigcup_{w \in M} \mathrm{Min}(\uparrow w \cap \uparrow \pi_1 \circ h \circ f(w)) \\
&= \bigcup_{w \in M} \uparrow \mathrm{Min}(\uparrow w \cap \uparrow \pi_1 \circ h \circ f(w)) \\
&= \bigcup_{w \in M} (\uparrow w \cap \uparrow \pi_1 \circ h \circ f(w))
\end{aligned}$$

The same applies to (N, g), so we just need to show that

$$\bigcup_{z \in N} (\uparrow z \cap \uparrow \pi_1 \circ h \circ g(z)) \subseteq \bigcup_{w \in M} (\uparrow w \cap \uparrow \pi_1 \circ h \circ f(w))$$

For each $z \in N$, there is a $w \in M$ that is below z. In this case, $\uparrow z \subseteq \uparrow w$ and

$$\uparrow(\pi_1 \circ h \circ g(z)) \subseteq \uparrow(\pi_1 \circ h \circ f(w))$$

since $g(z) \geq f(w)$ and h is monotone. Thus,

$$(\uparrow z \cap \uparrow(\pi_1 \circ h \circ g(z))) \subseteq (\uparrow w \cap \uparrow(\pi_1 \circ h \circ f(w)))$$

Now we check the functions. For $w \in (\pi_1 \circ h^\dagger(M, f))$ and $z \in (\pi_1 \circ h^\dagger(N, g))$ with $w \leq z$,

we must show that $(\pi_2 \circ h^\dagger(M,f))(w) \leq (\pi_2 \circ h^\dagger(N,g))(z)$.

Let $\pi_M(w)$ equal the unique word in M below w. Because $w \leq z$ and $M \sqsubseteq_{EM} N$, $\pi_M(w) \leq \pi_N(z)$, and $f \circ \pi_M(w) \leq g \circ \pi_N(z)$. Since h is monotone, $h \circ f \circ \pi_M(w) \leq h \circ g \circ \pi_N(z)$.

Again, $w \leq z$, so $\pi_{\pi_1 \circ h \circ f \circ \pi_M(w)}(w) \leq \pi_{\pi_1 \circ h \circ g \circ \pi_N(z)}(z)$, and we have

$$\begin{aligned}(\pi_2 \circ h^\dagger(M,f))(w) &= (\pi_2 \circ h \circ f \circ \pi_M(w))(\pi_{\pi_1 \circ h \circ f \circ \pi_M(w)}(w)) \\ &\leq (\pi_2 \circ h \circ g \circ \pi_N(z))(\pi_{\pi_1 \circ h \circ g \circ \pi_N(z)}(z)) \\ &= (\pi_2 \circ h^\dagger(N,g))(z)\end{aligned}$$

■

Theorem 2.34. *$h^\dagger$ is Scott continuous.*

Proof. It must shown that $h^\dagger$ preserves directed suprema. Let $\{(M_i, f_i)\}$ be a directed family in $RC(D)$. We begin by considering the first component of $h^\dagger$:

$$\begin{aligned}\bigsqcup_i \pi_1 \circ h^\dagger(M_i,f_i) &= \bigsqcup_i \bigcup_{w \in M_i} \mathrm{Min}(\uparrow w \cap \uparrow \pi_1 \circ h \circ f_i(w)) && (2.1)\\ &= \mathrm{Min} \bigcap_i \uparrow \bigcup_{w \in M_i} \mathrm{Min}(\uparrow w \cap \uparrow \pi_1 \circ h \circ f_i(w)) && (2.2)\\ &= \mathrm{Min} \bigcap_i \bigcup_{w \in M_i} (\uparrow w \cap \uparrow \pi_1 \circ h \circ f_i(w)) && (2.3)\\ &= \mathrm{Min} \bigcup_{w \in \sqcup_i M_i} \bigcap_i \uparrow w \cap \uparrow \pi_1 \circ h \circ f_i \circ \pi_{M_i}(w)) && (2.4)\\ &= \mathrm{Min} \bigcup_{w \in \sqcup_i M_i} \uparrow w \ \cap \bigcap_i \uparrow \pi_1 \circ h \circ f_i \circ \pi_{M_i}(w)) && (2.5)\\ &= \mathrm{Min} \bigcup_{w \in \sqcup_i M_i} \uparrow w \cap \uparrow \mathrm{Min} \bigcap_i \uparrow \pi_1 \circ h \circ f_i \circ \pi_{M_i}(w)) && (2.6)\\ &= \mathrm{Min} \bigcup_{w \in \sqcup_i M_i} \uparrow w \cap \uparrow \bigsqcup_i \pi_1 \circ h \circ f_i \circ \pi_{M_i}(w)) && (2.7)\\ &= \bigcup_{w \in \sqcup_i M_i} \mathrm{Min} \uparrow w \cap \uparrow \pi_1 \circ h \circ \bigsqcup_i f_i \circ \pi_{M_i}(w)) \\ &= \bigcup_{w \in \sqcup_i M_i} \mathrm{Min} \uparrow w \cap \uparrow \pi_1 \circ h \circ (\bigsqcup_i f_i)(w)) \\ &= \pi_1 \circ h^\dagger(\bigsqcup_i M_i, \bigsqcup_i f_i)\end{aligned}$$

The equations (2.1)=(2.2) and (2.6)=(2.7) stem from Fact 2.17. Equations (2.2)=(2.3) and (2.5)=(2.6)

hold since for an upper set with minimal elements, taking the upper set of those minimal elements results in the original upper set (there are no infinitely decreasing chains, so every element is above a minimal element).

Finally, for the equality (2.3)=(2.4), first note that the unions are all disjoint. The equality holds even without taking the minimal elements of both sides. If $z \in \bigcap_i \bigcup_{w \in M_i} (\uparrow w \cap \uparrow \pi_1 \circ h \circ f_i(w))$, then z is above some w_i for every M_i, and therefore, it is above some $v \in \sqcup_i M_i$. It is also above $\pi_1 \circ h \circ f_i(w_i) = \pi_1 \circ h \circ f_i \circ \pi_{M_i}(v)$ for every M_i. Thus, $z \in \bigcup_{w \in \sqcup_i M_i} \bigcap_i \uparrow w \cap \uparrow \pi_1 \circ h \circ f_i \circ \pi_{M_i}(w))$. Conversely, if z is in (2.4), then it is above some $v \in \sqcup_i M_i$ and it is above $\pi_1 \circ h \circ f_i \circ \pi_{M_i}(v)$ for all M_i. Each M_i has a $w_i \leq v$, so z is above each w_i. Again, $\pi_1 \circ h \circ f_i \circ \pi_{M_i}(v) = \pi_1 \circ h \circ f_i(w_i)$, so z is above each of these, placing it in (2.3).

Now we check the second component of $h^\dagger$:

$$\begin{aligned}
\bigsqcup_i (\pi_2 \circ h^\dagger(M_i, f_i))(w) &= \bigsqcup_i (\pi_2 \circ h \circ f_i \circ \pi_{M_i}(w))(\pi_{\pi_1 \circ h \circ f_i \circ \pi_{M_i}}(w)) \\
&= \pi_2 \circ h \circ \bigsqcup_i f_i \circ \pi_{M_i}(w))(\pi_{\pi_1 \circ h \circ f_i \circ \pi_{M_i}}(w)) \\
&= (\pi_2 \circ h^\dagger(\sqcup_i M_i, \sqcup_i f_i))(w)
\end{aligned}$$

■

As can be seen in the above proof, the notation can get cumbersome due to the many projections. Here we introduce a more concise notation, based on the fact that any continuous function $f : M \to D$ can easily be extended to the upper set of M by $f \circ \pi_M : \uparrow M \to D$.

Notation For $f : M \to D$ and $h : D \to RC(E)$, define the function $h_1^f : \uparrow M \to FAC(\{0,1\}^\infty)$ by $h_1^f(w) = \pi_1 \circ h \circ f(\pi_M(w))$, and define $h_2^f(w) = \pi_2 \circ h \circ f(\pi_M(w))$. Furthermore, for any z in $\uparrow(h_1^f(w))$, let $(h_2^f(w))(z) = (h_2^f(w))(\pi_{(h_1^f(w))}(z))$.

Note that if $w \geq w'$, then $h_1^f(w) = h_1^f(w')$ and $h_2^f(w) = h_2^f(w')$. Also, if $z \geq z'$, then $(h_2^f(w))(z) = (h_2^f(w))(z')$.

Definition of $h^\dagger$ with notation

$$\pi_1 \circ h^\dagger(M, f) = \bigcup_{w \in M} \mathrm{Min}(\uparrow w \cap \uparrow(h_1^f(w)))$$

$$((\pi_2 \circ h^\dagger)(M, f))(z) = (h_2^f(z))(z)$$

This lemma will be useful in proving that the monad laws hold.

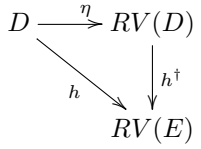
Lemma 2.35. *If A is an upper set in $\{0,1\}^\infty$, then $\uparrow\mathrm{Min}(A) = A$.*

Proof. $\mathrm{Min}(A) \subseteq A$, and since A is an upper set, $\uparrow\mathrm{Min}(A) \subseteq A$. Conversely, there are no infinite descending chains in $\{0,1\}^\infty$, so every word in A is above some minimal element of A. Thus, $A \subseteq \uparrow\mathrm{Min}(A)$. ■

Theorem 2.36. *The functor RC forms a monad.*

Proof. Now the three monad laws are shown to hold.

$[h^\dagger \circ \eta = h]$



$$\begin{aligned}
\pi_1 \circ h^\dagger \circ \eta(d) &= \pi_1 \circ h^\dagger(\delta_\epsilon, \chi_d)) \\
&= \mathrm{Min}(\uparrow\epsilon \cap \uparrow(h_1^{\chi_d}(\epsilon))) \\
&= \mathrm{Min}(\uparrow(h_1^{\chi_d}(\epsilon))) \\
&= h_1^{\chi_d}(\epsilon) \\
&= \pi_1 \circ h \circ \chi_d(\epsilon) \\
&= \pi_1 \circ h(d)
\end{aligned}$$

$$\begin{aligned}
(\pi_2 \circ h^\dagger \circ \eta(d))(z) &= (\pi_2 \circ h^\dagger(\delta_\epsilon, \chi_d))(z) \\
&= (h_2^{\chi_d}(z))(z) \\
&= (\pi_2 \circ h \circ \chi_d(z))(z) \\
&= (\pi_2 \circ h(d))(z).
\end{aligned}$$

$[\boldsymbol{\eta^\dagger = \mathrm{id}}]$

$$\begin{aligned}
\pi_1 \circ \eta^\dagger(M, f) &= \bigcup_{w \in M} \mathrm{Min}(\uparrow w \cap \uparrow(\eta_1^f(w))) \\
&= \bigcup_{w \in M} \mathrm{Min}(\uparrow w \cap \uparrow \epsilon) \\
&= \bigcup_{w \in M} \mathrm{Min}(\uparrow w) \\
&= \bigcup_{w \in M} w \\
&= M
\end{aligned}$$

$$\begin{aligned}
(\pi_2 \circ \eta^\dagger)(M, f)(z) &= \eta_2^f(z)(z) \\
&= \chi_{f(z)}(z) \\
&= f(z)
\end{aligned}$$

$[\boldsymbol{k^\dagger \circ h^\dagger = (h^\dagger \circ h)^\dagger}]$

For any function $h : D \to RC(E)$ and $(M, f) \in RC(D)$, the first component of the Kleisli extension, $h^\dagger(M, f)$, will be an antichain at least as big as M. For any $w \in M$, the amount that the Kleisli extension grows above w is determined solely by $h_1^f(w)$. To show that the equality, $k^\dagger \circ h^\dagger = (h^\dagger \circ h)^\dagger$, holds for the antichains, we can show that the antichains are equal above any arbitrary w in M.

Let w be any word in M.

$$\uparrow w \cap (\pi_1 \circ k^\dagger \circ h^\dagger(M, f)) = \bigcup_{z \in \mathrm{Min}(\uparrow w \ \cap \ \uparrow(h_1^f(w)))} \mathrm{Min}(\uparrow z \cap \uparrow(k_1^{h_2^f(z)}(z)))$$

$$\begin{aligned}
\uparrow w \cap (\pi_1 \circ (k^\dagger \circ h)^\dagger(M,f)) &= \mathrm{Min}(\uparrow w \cap \uparrow(\pi_1 \circ k^\dagger \circ h \circ f(w))) \\
&= \mathrm{Min}(\uparrow w \cap \uparrow \bigcup_{z \in h_1^f(w)} \mathrm{Min}(\uparrow z \cap \uparrow(k_1^{h_2^f(w)}(z)))) \\
&= \mathrm{Min}(\uparrow w \cap \bigcup_{z \in h_1^f(w)} \uparrow \mathrm{Min}(\uparrow z \cap \uparrow(k_1^{h_2^f(w)}(z)))) \\
&= \mathrm{Min}(\uparrow w \cap \bigcup_{z \in h_1^f(w)} (\uparrow z \cap \uparrow(k_1^{h_2^f(w)}(z)))) \\
&= \mathrm{Min}(\bigcup_{z \in h_1^f(w)} (\uparrow w \cap \uparrow z \cap \uparrow(k_1^{h_2^f(w)}(z))))
\end{aligned}$$

Now suppose w is not in $\downarrow h_1^f(w)$. Then, for $k^\dagger \circ h^\dagger$, the only z to consider is w. Thus:

$$\uparrow w \cap (\pi_1 \circ k^\dagger \circ h^\dagger(M,f)) = \mathrm{Min}(\uparrow w \cap \uparrow(k_1^{h_2^f(w)}(w)))$$

For $(k^\dagger \circ h)^\dagger$, in $\bigcup_{z \in (h_1^f(w))}$, we only need to consider $z \in h_1^f(w) \cap \downarrow w$, which only contains $\pi_{h_1^f(w)}(w)$. This z is smaller than w, so $\uparrow w \cap \uparrow z = \uparrow w$. Also, $k_1^{h_2^f(w)}(z) = k_1^{h_2^f(w)}(w)$. Thus we get:

$$\uparrow w \cap (\pi_1 \circ (k^\dagger \circ h)^\dagger(M,f)) = \mathrm{Min}(\uparrow w \cap \uparrow(k_1^{h_2^f(w)}(w)))$$

Now suppose w is in $\downarrow h_1^f(w)$. Then, for $k^\dagger \circ h^\dagger$, we have to consider z in $\uparrow w \cap h_1^f(w)$. Thus:

$$\uparrow w \cap (\pi_1 \circ k^\dagger \circ h^\dagger(M,f)) = \bigcup_{z \in \uparrow w \cap (h_1^f(w))} \mathrm{Min}(\uparrow z \cap \uparrow(k_1^{h_2^f(z)}(z)))$$

For $(k^\dagger \circ h)^\dagger$, in $\bigcup_{z \in h_1^f(w)}$, we only need to consider $z \in h_1^f(w) \cap \uparrow w$. Since each z is above w, $\uparrow z \cap \uparrow w = \uparrow z$. Thus we get:

$$\begin{aligned}
\uparrow w \cap (\pi_1 \circ (k^\dagger \circ h)^\dagger(M,f)) &= \mathrm{Min}(\bigcup_{z \in \uparrow w \cap (h_1^f(w))} (\uparrow z \cap \uparrow(k_1^{h_2^f(w)}(z)))) \\
&= \bigcup_{z \in \uparrow w \cap (h_1^f(w))} \mathrm{Min}(\uparrow z \cap \uparrow(k_1^{h_2^f(z)}(z)))
\end{aligned}$$

Finally, we have to check the functions.

$$(\pi_2 \circ k^\dagger \circ h^\dagger(M, f))(z) = (k_2^{h_2^f(z)}(z))(z)$$

$$\begin{aligned}(\pi_2 \circ (k^\dagger \circ h)^\dagger(M, f))(z) &= ((k^\dagger \circ h)_2^f(z))(z) \\ &= (\pi_2 \circ k^\dagger \circ h \circ f(\pi_M(z)))(z) \\ &= (\pi_2 \circ k^\dagger(h_1^f(z), h_2^f(z)))(z) \\ &= (k_2^{h_2^f(z)}(z))(z)\end{aligned}$$

■

Here, we created a monad by defining both the unit and Kleisli extension. Another characterization of a monad involves the multiplication of the monad, which is a natural transformation $\mu : RC^2 \to RC$. Given the Kleisli extension and the identity function $\mathsf{id} : RC(D) \to RC(D)$, the multiplication is simply the Kleisli extension of the identity function.

Let (M, f) be in $RC^2(D)$. Then f is a function from M to $RC(D)$. We can write f as (f_1, f_2), where for a w in M, f_1 gives a full antichain, and f_2 gives a function from that antichain to D. Now the multiplication is defined as:

$$\mu(M, f) = (\bigcup_{w \in M} \mathrm{Min}(\uparrow w \cap \uparrow f_1(w)), w \mapsto (f_2(w))(w))$$

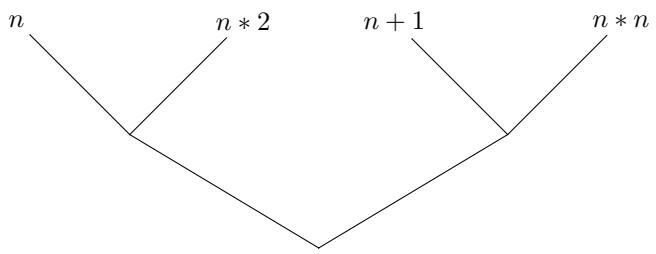
where f_2 and $f_2(w)$ may have to be extended upward to be well defined.

2.4 More About the Kleisli Extension

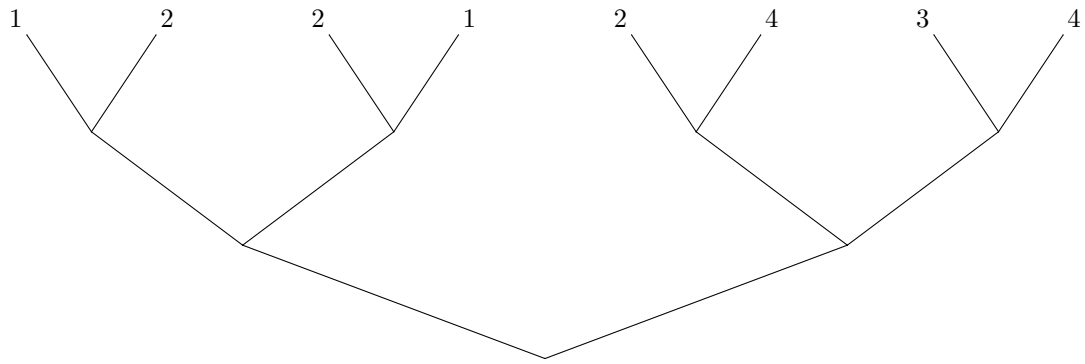
2.4.1 Dependent Choice

Consider a function $f : \mathbb{N} \to RC(\mathbb{N})$ that takes a natural number, n, and flips a coin twice to determine what to do with that number. If two tails are flipped, the number is left alone. If two heads are flipped, the number is squared. If tails, then heads appear, then the number is doubled, and if heads, then tails appear, the number is incremented by 1. Thus, for a given n, $f(n)$ is equal

to the following tree:

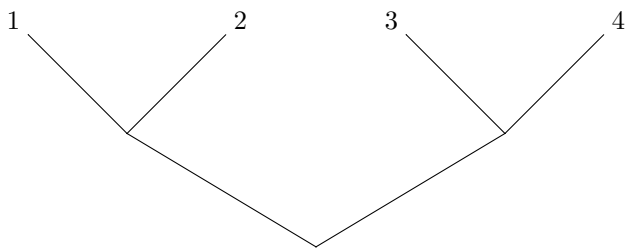


Now if a Klesili extension is applied to f, it can take as an argument something of type $RC(\mathbb{N})$. Giving it (ϵ, χ_n), where no coin flips are made, will have the same behavior as $f(n)$ (this is the first monad law). However, suppose there is a coin flip to choose n, so that $n = 1$ if the coin is tails and $n = 2$ if the coin is heads. Then we apply f to this random choice of n, so we have a random choice of actions to perform on a random choice of n. If the choices are made independently, then the resulting random choice would look like:



However, this is not how our Kleisli extension behaves. Instead, f assumes that its first coin flip will be the same as the one used to choose n. If 1 is chosen, then the result would be $f(1 \mid \text{first flip is tails})$, so only one more coin flip would be made to either leave 1 alone or double it. Similarly, if 2 is chosen, the result would be $f(2 \mid \text{first flip is heads})$, so the one remaining coin flip

would determine if 2 is incremented by one or squared. The resulting tree would be as follows:

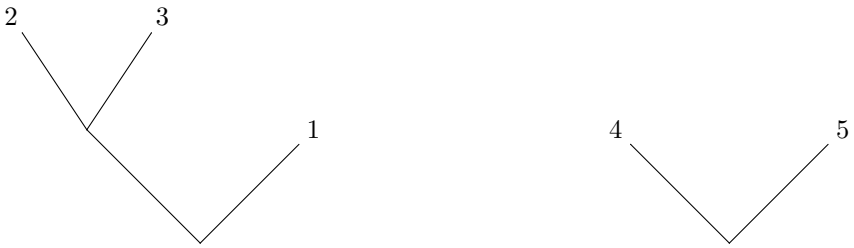


If probabilities are brought into the picture, then the difference between this Kleisli extension and the first Kleisli extension would be that in the first, the probabilistic choices would be independent, but in our Kleisli extension, some probabilistic choices would be dependent on other prior choices.

2.4.2 Lifting of Binary Operations

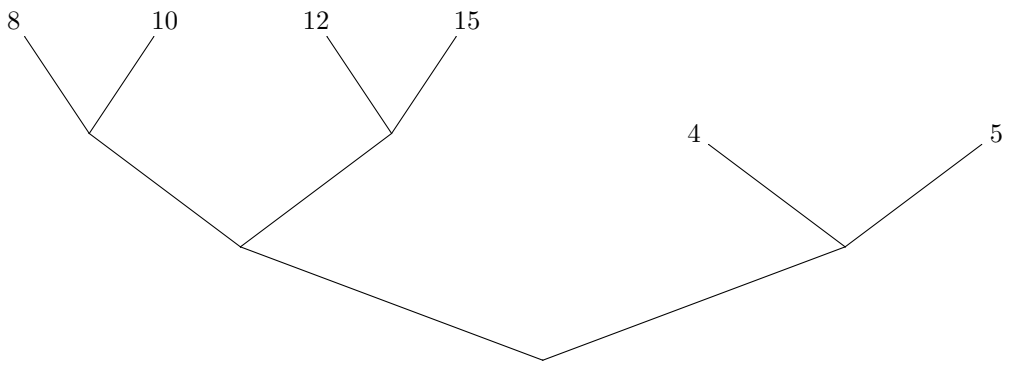
The Kleisli extension of a monad T can be hard to think about intuitively since we normally do not work with functions from D to $T(E)$. However, the Kleisli extension is important in lifting binary operations on the underlying structures to binary operations on the monadic structures. If we have a binary operation $* : D \times E \to F$, the Kleisli extension lifts this to the binary operation $*^\dagger : T(D) \times T(E) \to T(F)$. This is achieved by setting $*^\dagger = (\lambda a.T(\lambda b.a * b))^\dagger$.

Now consider the monad of randomized choice, where our underlying domain is the natural numbers (with the usual ordering, with infinity on top). How should we lift a binary operation like multiplication to operate on a random choice of natural numbers? Consider two trees:

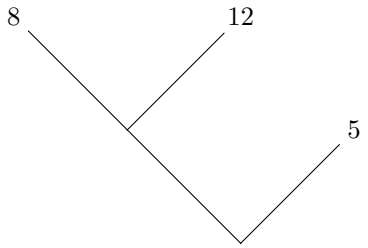


The first attempt at the Kleisli extension would perform the random choices of numbers

sequentially and multiply the two random numbers. The resulting tree would be as follows:

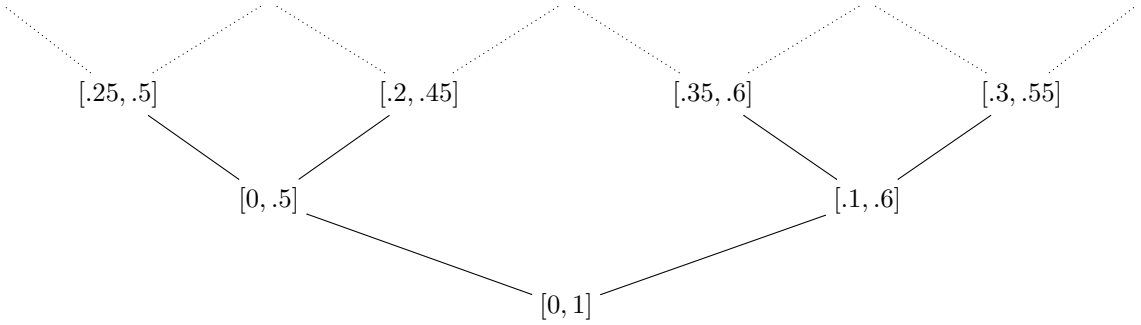


The Kleisli extension presented here behaves differently. Instead of making the random choices sequentially, it makes them concurrently. When a random bit is generated, it gets sent to both input trees. In the examples trees above, if the first bit is a 1, then the first tree will choose 1, and the second tree will choose 5. If the first bit is a 0, then the second tree will choose 4, but the first tree will still need another bit to choose between 2 and 3. The resulting tree is as follows:



It may seem odd that not all random choices appear in the resulting tree (since a random choice is applied to both arguments simultaneously), but for randomized algorithms, this is not a problem. Randomized algorithms are used to find a specific output that should not change based on the random choices made. For example, we may be interested in finding a specific real number in a known interval, such as a root of a polynomial. We may have two different ways of shrinking this interval around the desired number, such as the bisection method. At each step, we can randomly choose which method to apply. We may not be able to find the exact number in finitely many steps, but the resulting intervals should converge to that number. Therefore, our computation can be

modeled with an infinite tree, where each value is in the interval domain, defined in Example 1.23.



Now suppose that there are real numbers whose sum we wish to compute. If $x \in [a, b]$, and $y \in [c, d]$, then we know that $x + y \in [a + c, b + d]$. Therefore, we can use the Kleisli extension to lift the addition of real numbers to the addition of our trees of intervals. If we tried to perform our random algorithms sequentially for each real number, it will not work since the first algorithm never ends. We will keep pinning down the first number, but the information about the second real number will remain at the initial interval containing it. However, as stated above, the Kleisli extension presented here performs the random choices concurrently, not sequentially. Therefore, the intervals surrounding both numbers will become smaller, allowing our intervals for the sum to converge to sum of the real numbers.

2.5 The Miller-Rabin Algorithm

One of most well known randomized algorithms is the Miller-Rabin primality test. To test whether a given number n is prime, a random number is chosen between 2 and $n - 2$. Tests using modular arithmetic are performed with this random number before determining whether the given number is composite or probably prime. The test can be run in polynomial time, but it has a possible one-sided error, putting primality testing in the complexity class of randomized polynomial time (RP). A test on a prime number will always return "probably prime", but sometimes, a test on a composite number will also return "probably prime". Thus, if the test returns "composite", there is no chance for error, but a return of "probably prime" always has a chance of error. For a composite number, at most $\frac{1}{4}$ of the possible random choices between 2 and $n - 2$ will result in the test returning "probably prime". To minimize the error probability, we can repeat the test (choosing a new random number) only when the test returns "probably prime". Running the test m times results in an error probability of at most $\frac{1}{4^m}$.

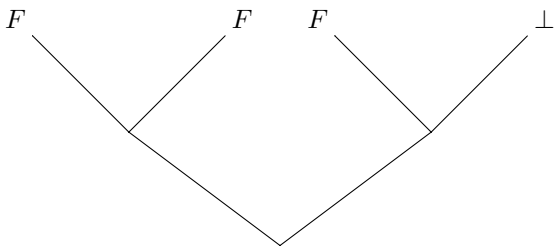


Figure 2.6: One possible iteration of a simplified Miller-Rabin test on a composite number.

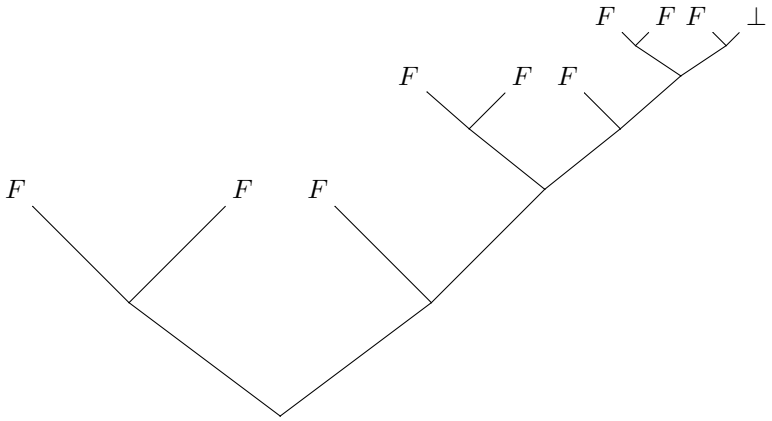


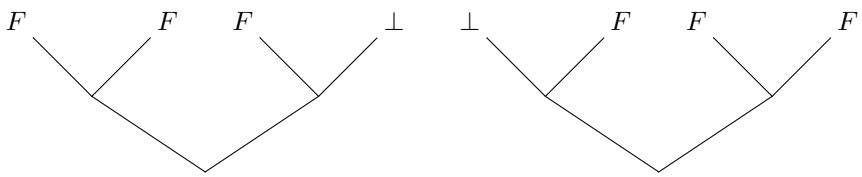
Figure 2.7: Three iterations of a hypothetical Miller-Rabin test.

Figure 2.6 shows the possible outcomes of a hypothetical Miller-Rabin test on a composite number. For simplicity, it is assumed that a random number between 2 and $n-2$ can be properly chosen using just two coin flips. Each coin flip is represented by a branching of the binary tree. The top of the tree is labeled with the return values of the test using the random numbers chosen by the resulting outcome of two coin flips. If the test returns "composite", an "F" is used whereas "$\perp$" denotes "probably prime". A "T" is not used since a Miller-Rabin test never confirms that a number is prime. If we wish to minimize the error probability, we can choose to run the test again, which will expand the tree wherever a "$\perp$" is found.

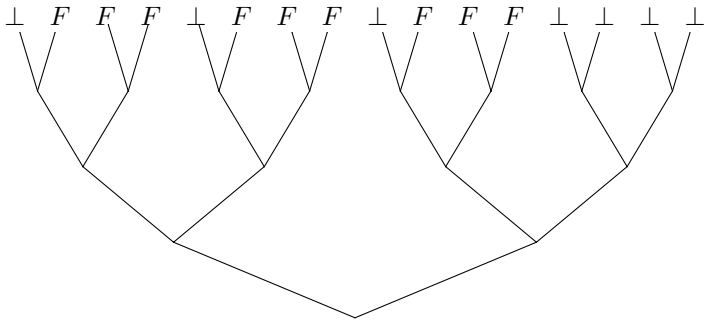
Figure 2.7 shows the possible outcomes of using Miller-Rabin a maximum of three times on the same composite number. This can be extended similarly to an infinite tree with a zero probability of error.

Suppose that we have a Miller-Rabin test performed on two composite numbers with the

following possible outcomes:



How should the binary operation **or** be lifted? It may seem natural to perform the two tests one after the other, resulting in:

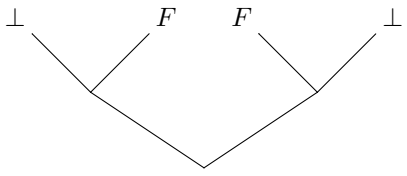


The probability of error in this case is $\frac{7}{16}$, assuming we use a fair coin. However, this method has two main flaws.

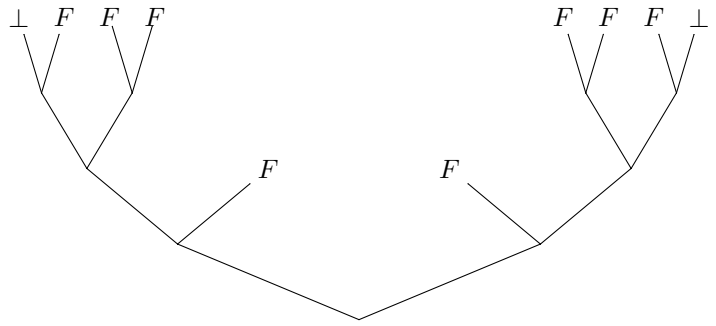
1. How do we handle the infinite case? If the first random test can use infinitely many coin flips, then the second test will never even start.

2. The Kleisli extension that results in this behavior is not monotone. Therefore, it does not form a monad in a category we want.

Instead, consider feeding the result of each coin flip to both tests concurrently. For two coin flips, our example would look like:



To properly compare it with the sequential case, we should use the same maximum number of coin flips. Feeding all four coin flips to both Miller-Rabin tests results in:



which only has an error probability of $\frac{1}{8}$. If the error possibility for each number had coincided, then the error probability would have been smaller, $\frac{1}{16}$.

In fact, this can be extended to testing primality for a set of m integers. If we get enough random bits to perform the Miller-Rabin test n times, we can use the same bits to test all m numbers. Then the probability that at least one error will occur is $m(\frac{1}{4})^n$. For each integer, at most $(\frac{1}{4})^n$ of the possible random choices will result in an error. The worst case scenario is when the error chances are all disjoint, resulting in $m(\frac{1}{4})^n$. This idea is used in [35] to search for the smallest prime bigger than some integer m.

Thus, for randomized algorithms, our Kleisli extension allows us to use random bits more efficiently. Getting random bits can be an expensive or slow process, so it is beneficial to minimize the error probability using fewer random bits.

2.6 Relation to Scott's Stochastic Lambda Calculus

Dana Scott developed an operational semantics of the lambda calculus using the power set of natural numbers, $\mathcal{P}(\mathbb{N})$. As terms of the lambda calculus, elements of $\mathcal{P}(\mathbb{N})$ can be applied to one another and λ-abstraction is achieved through the use of enumerations similar to Gödel numbering.

Scott then added randomness to his model, resulting in his stochastic lambda calculus [36]. He does this by adding random variables.

Definition 2.37. A *random variable* in Scott's $\mathcal{P}(\mathbb{N})$ model is a function $\mathsf{X} : [0,1] \to \mathcal{P}(\mathbb{N})$ where $\{t \in [0,1] | n \in \mathsf{X}(t)\}$ is Lebesgue measurable for all n in $\mathcal{P}(\mathbb{N})$.

This is similar to the monad of random choice presented in this paper. We start with a base domain D, which could be $\mathcal{P}(\mathbb{N})$, and then have a function from a full antichain of $\{0,1\}^\infty$, M, into D. We can really treat this as a function from the Cantor space, $\{0,1\}^\omega$ to D. Since M is a full antichain, $M \sqsubseteq_{EM} \{0,1\}^\omega$. Thus we can extend f to $\overline{f} : \{0,1\}^\omega \to D$, where $\overline{f}(w) = f \circ \pi_M(w)$.

Now that random variables are added to the lambda calculus, there must be a way to define application of one random variable to another. In a sense, this is lifting the application operation from $\mathcal{P}(\mathbb{N})$ to $[0,1] \to \mathcal{P}(\mathbb{N})$, which, as stated above, is the role of the Kleisli extension of the monad. Scott defines the application as follows:

Definition 2.38. Given two random variables $\mathsf{X}, \mathsf{Y} : [0,1] \to \mathcal{P}(\mathbb{N})$, the *application operation* is defined by

$$\mathsf{X}(\mathsf{Y})(t) = \mathsf{X}(t)(\mathsf{Y}(t))$$

These random variables can be thought of as using an oracle that randomly gives a element of $[0,1]$, and then the function of the random variable uses this number to output an element of $\mathcal{P}(\mathbb{N})$. Notice that in the above definition for application, both random variables receive the same t. Thus, the oracle is consulted only once instead of giving a different random number to each. This exactly mimics the concurrent operation of the Kleisli extension. But instead of an oracle giving an entire real number at once (which has infinite information), the oracle gives one bit at a time.

Chapter 3

Distributive Laws and Variations on the Monad

We can model the addition of computational effects to a programming language by using monads. If multiple effects are needed, then the corresponding monads can be composed to obtain another monad, but only if there is a distributive law between the two monads. Our random choice monad gives a way to model randomized algorithms in a programming language. If we also want to model nondeterminism, then we should combine the random choice monad with one of the probabilistic powerdomains. To do this, we must first show that there is a distributive law.

3.1 Beck's Distributive Law

For two monads, S and T, over the same category, the functor TS is not necessarily a monad. According to Beck's Theorem [5], the composition of two monads, S and T, is a monad if and only if there is a distributive law between them. A distributive law consists of a natural transformation $\lambda : ST \to TS$ that satisfies the following equations:

1. $\lambda \circ S\eta^T = \eta^T S$
2. $\lambda \circ \eta^S T = T\eta^S$
3. $\lambda \circ S\mu^T = \mu^T S \circ T\lambda \circ \lambda T$
4. $\lambda \circ \mu^S T = T\mu^S \circ \lambda S \circ S\lambda$

For the composition of two monads, TS, to be a monad, the multiplication $\mu^{TS} : TSTS \to TS$ must be defined. Given $TSTS$ we can use the natural transformation λ to get $TTSS$. Then the multiplications of T and S can each be used to get TS. The above equations ensure that the multiplication defined in this manner will satisfy the monad laws.

3.2 Distributive Law With the Lower Powerdomain

We now show that the random choice monad enjoys a distributive law with the lower powerdomain in the category BCD. The lower, or Hoare, powerdomain was defined in Section 1.2.5. For a domain X, it consists of the nonempty Scott closed sets of X, denoted $\Gamma_0(X)$. The unit of the lower powerdomain, $\eta : X \to \Gamma_0(X)$ is defined by:

$$\eta(x) = \downarrow x$$

The multiplication, $\mu : \Gamma_0^2(X) \to \Gamma_0(X)$ is defined as:

$$\mu(S) = \overline{\bigcup_{U \in S} U}$$

Let a be a morphism from X to Y. For a Scott closed set, U, of X, $\Gamma_0(a)(U) = \overline{\{a(u) | u \in U\}}$. Objects of $RC \circ \Gamma_0(X)$ are random variables (M, f), with $f : M \to \Gamma_0(X)$. Objects of $\Gamma_0 \circ RC(X)$ are Scott closed sets of random variables on X. For $(M, f) \in RC \circ \Gamma_0(X)$,

$$\begin{aligned} RC \circ \Gamma_0(a)(M, f) &= (M, \Gamma_0(a) \circ f) \\ &= (M, w \mapsto \overline{\bigcup_{x \in f(w)} a(x)}) \end{aligned}$$

For a Scott closed set, U, of $\Gamma_0 \circ RC(X)$,

$$\Gamma_0 \circ RC(a)(U) = \overline{\{(M, a \circ f) \mid (M, f) \in U\}}$$

To simplify the following diagrams, let T be the random choice monad and let S be the lower powerdomain. Now suppose $(M, f) \in TS(X)$, so that f is a function from M to $\Gamma_0(X)$. Define the natural transformation $\lambda : TS \to ST$ so that for an object (M, f) in $TS(X)$:

$$\lambda_X(M, f) = \downarrow\{(M, g) \mid g(w) \in f(w), \forall w \in M\}$$

This set is Scott closed because each $f(w)$ is Scott closed as we now show: For any directed set $D = \{N_i, h_i\}$ in $\lambda(M, f)$, there is a directed set $E = \{M, \overline{h_i}\}$, where $\overline{h_i}(w) = h_i \circ \pi_{N_i}(w)$, and $\overline{h_i}(w)$ is contained in $f(w)$ for each i. $\bigsqcup D \sqsubseteq \bigsqcup E = (M, w \mapsto \bigsqcup_i \overline{h_i}(w))$. Since $\overline{h_i}(w)$ is in $f(w)$ for each i and $f(w)$ is closed, $\bigsqcup_i \overline{h_i}(w)$ is in $f(w)$. Therefore, $\bigsqcup E$ is in $\lambda_X(M, f)$. $\bigsqcup D$ is smaller, so it must also be in $\lambda_X(M, f)$.

Proposition 3.1. *For any bounded complete domain, X, $\lambda_X : TS(X) \to ST(X)$ is monotone.*

Proof. If $(N, h) \sqsubseteq (M, f)$ then $N \subseteq \downarrow M$ and for any $z \in N$ and $w \in M$ with $z \leq w$, $h(z) \subseteq f(w)$. We must show that

$$\lambda_X(N, h) = \downarrow\{(N, g) \mid g(z) \in h(z), \forall z \in N\} \subseteq \downarrow\{(M, g) \mid g(w) \in f(w), \forall w \in M\} = \lambda_X(M, f)$$

Any (N, g) can be extended to $(M, \overline{g})$ with $\overline{g}(w) = g \circ \pi_N(w)$. For each w, $\overline{g}(w) = g(z)$ for some $z \leq w$, so $\overline{g}(w) \in h(z)$ and since $h(z) \subseteq f(w)$, $\overline{g}(w) \in f(w)$. Therefore, $(M, \overline{g}) \in \lambda_X(M, f)$ and $(N, g) \sqsubseteq (M, \overline{g})$, so $(N, g) \in \lambda_X(M, f)$. ■

Proposition 3.2. *For any bounded complete domain, X, $\lambda_X : TS(X) \to ST(X)$ is Scott continuous.*

Proof. For a directed set $\{M_i, f_i\}$ in X, its supremum is $(\bigsqcup_i M_i, w \mapsto \overline{\bigcup_i f_i \circ \pi_{M_i}(w)})$. Applying λ_X results in

$$\downarrow\{(\bigsqcup_i M_i, g) \mid g(w) \in \overline{\bigcup_i f_i \circ \pi_{M_i}(w)}\} = \overline{\{(\bigsqcup_i M_i, g) \mid g(w) \in \bigcup_i f_i \circ \pi_{M_i}(w)\}} \tag{1}$$

We must show that this is equal to

$$\overline{\bigcup_i \downarrow\{(M_i, g) \mid g(w) \in f_i(w)\}} = \overline{\bigcup_i \{(M_i, g) \mid g(w) \in f_i(w)\}} \tag{2}$$

It is clear that (2) $\subseteq$ (1) since any (M_i, g) can be extended to $(\bigsqcup_i M_i, \overline{g})$, where $\overline{g}(w) = g \circ \pi_{M_i}(w)$. Thus, we need to show that (1) $\subseteq$ (2).

Now consider some $(\bigsqcup_i M_i, g)$ in (1). From Corollary 2.28, $(\bigsqcup_i M_i, g)$ is the supremum of its projections to finite antichains, $\pi_n(\bigsqcup_i M_i, g)$. If we can show that $\pi_n(\bigsqcup_i M_i, g)$ is in (2) for all n, then $(\bigsqcup_i M_i, g)$ must also be in (2) since (2) is Scott closed. For a fixed n, $\pi_n(\bigsqcup_i M_i)$ is way below $\bigsqcup_i M_i$ so there is some j such that $\pi_n(\bigsqcup_i M_i) \sqsubseteq M_j$. We can assume that all M_i in our directed set are above M_j. For any w in $\bigsqcup_i M_i$, there is a k such that $g(w) \in f_k \circ \pi_{M_k}(w)$. For any z in $\pi_n(\bigsqcup_i M_i)$, $\pi_n(g)(z)$ is below $g(w)$ for any w above z. Thus there is a k such that $\pi_n(g)(z) \in f_k(w)$ for any w above z. There are finitely many such words z that each have a corresponding k. Since $\{M_i, f_i\}$ is directed, there is a (M_p, f_p) above all such (M_k, f_k). Now $\pi_n(\bigsqcup_i M_i)$ is below M_p and for all z in $\pi_n(\bigsqcup_i M_i)$ and w in M_p with $z \leq w$, $\pi_n(g)(z) \in f_p(w)$. Therefore, we can extend $\pi_n(\bigsqcup_i M_i, g)$ to $(M_p, \overline{\pi_n(g)})$, where $\overline{\pi_n(g)}(w) \in f_p(w)$, so it is in (2). ■

In proving that the following diagrams commute, we can assume that for each (M, f), the antichain is finite. If not, $(M, f) = \bigsqcup_n \pi_n(M, f)$, by Corollary 2.28. Each $\pi_n(M, f)$ has a finite antichain, so if the equations hold for elements with finite antichains, then by continuity, they must hold for all elements. This is helpful since on finite antichains, all functions are continuous.

Proposition 3.3. *λ is a natural transformation.*

Proof. Suppose X and Y are bounded complete domains and $a : X \to Y$ is a Scott continuous function. For λ to be a natural transformation, the following diagram must commute:

$$\begin{array}{ccc} TS(X) & \xrightarrow{TS(a)} & TS(Y) \\ {\scriptstyle \lambda_X}\downarrow & & \downarrow{\scriptstyle \lambda_Y} \\ ST(X) & \xrightarrow[ST(a)]{} & ST(Y) \end{array}$$

We check this here:

$$\begin{aligned} \lambda_Y \circ TS(a)(M, f)) &= \lambda_Y(M, w \mapsto \overline{\bigcup_{x \in f(w)} a(x)}) \\ &= \downarrow\{(M, g) \mid g(w) \in \overline{\bigcup_{x \in f(w)} a(x)}, \forall w \in M\} \\ &= \overline{\{(M, g) \mid g(w) \in \bigcup_{x \in f(w)} a(x), \forall w \in M\}} \end{aligned} \tag{1}$$

$$\begin{aligned} ST(a) \circ \lambda_X(M, f) &= ST(a)(\downarrow\{(M, h) \mid h(w) \in f(w), \forall w \in M\}) \\ &= \overline{\{(N, a \circ j) \mid (N, j) \in \downarrow\{(M, h) \mid h(w) \in f(w), \forall w \in M\}\}} \\ &= \overline{\{(M, a \circ h) \mid h(w) \in f(w), \forall w \in M\}} \end{aligned} \tag{2}$$

(1) is a subset of (2) since for each g in (1) and $w \in M$, $g(w) = a(x_w)$ for some $x_w \in f(w)$. We can construct a corresponding h in (2) by setting $h(w) = x_w$. (2) is a subset of (1) since for each h in (2), we can construct the corresponding g as $g = a \circ h$. Therefore, (1) and (2) are equal. Also note that since all functions are continuous, we only have to take the closure at the very end. ■

Proposition 3.4. *In the category of bounded complete domains, there is a distributive law between the monad of random choice and the lower powerdomain, using the natural transformation λ.*

Proof. $[\boldsymbol{\lambda \circ T\eta^S = \eta^S T}]$

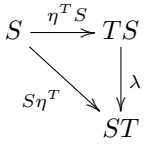
$$\begin{array}{ccc} T & \xrightarrow{T\eta^S} & TS \\ & {\scriptstyle \eta^S T}\searrow & \downarrow{\scriptstyle \lambda} \\ & & ST \end{array}$$

Let (M, f) be in $T(X)$.

$$\begin{aligned}\lambda_X \circ T\eta^S(M, f) &= \lambda_X(M, w \mapsto \downarrow f(w)) \\ &= \downarrow\{(M, g) \mid g(w) \in \downarrow f(w)\} \\ &= \downarrow\{(M, f)\}\end{aligned}$$

$$\eta^S T(M, f) = \downarrow\{(M, f)\}$$

$[\boldsymbol{\lambda \circ \eta^T S = S\eta^T}]$



Let U be a closed set in $S(X)$.

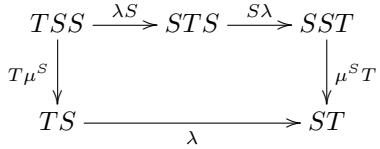
$$\begin{aligned}\lambda_X \circ \eta^T S(U) &= \lambda_X(\epsilon, \chi_U) \\ &= \downarrow\{(\epsilon, g) \mid g(\epsilon) \in U\} \\ &= \{(\epsilon, g) \mid g(\epsilon) \in U\}\end{aligned}$$

The last equality holds because U is a lower set.

$$\begin{aligned}S\eta^T(U) &= \overline{\{(\epsilon, g) | g(\epsilon) \in U\}} \\ &= \{(\epsilon, g) | g(\epsilon) \in U\}\end{aligned}$$

Here, the last equality holds since U is closed.

$[\boldsymbol{\lambda \circ T\mu^S = \mu^S T \circ S\lambda \circ \lambda_X S}]$



Let (M, f) be an element of $TSS(X)$ so that f is a function from M to $S^2(X)$.

$$\begin{aligned}\lambda_X \circ T\mu^S(M,f) &= \lambda_X(M, w \mapsto \overline{\bigcup_{U \in f(w)} U}) \\ &= \downarrow\{(M,h) \mid h(w) \in \overline{\bigcup_{U \in f(w)} U}\} \\ &= \overline{\{(M,h) \mid h(w) \in \bigcup_{U \in f(w)} U\}} \qquad (1)\end{aligned}$$

$$\begin{aligned}\mu^S T \circ S\lambda_X \circ \lambda_X S(M,f) &= \mu^S T \circ S\lambda_X(\downarrow\{(M,g) \mid g(w) \in f(w)\}) \\ &= \mu^S T(\overline{\{\lambda_X(M,g) \mid g(w) \in f(w)\}}) \\ &= \mu^S T(\overline{\{\{(M,h) \mid h(w) \in g(w)\} \mid g(w) \in f(w)\}}) \\ &= \overline{\{(M,h) \mid h(w) \in g(w), g(w) \in f(w)\}} \qquad (2)\end{aligned}$$

We show that (1) and (2) are equal before taking the closure. First we show (1) $\subseteq$ (2). For each h in (1) and $w \in M$, $h(w) \in U_w$ for some $U_w \in f(w)$. We can construct a corresponding g in (2) by setting $g(w) = U_w$. Now we show (2) $\subseteq$ (1). For each h in (2), $h(w) \in g(w)$. In (1), $h(w)$ has to be in at least one $U \in f(w)$ But $g(w) \in f(w)$, so the required U can taken as $g(w)$. Therefore, (1) and (2) are equal.

$[\boldsymbol{\lambda \circ \mu^T S = S\mu^T \circ \lambda T \circ T\lambda}]$

$$\begin{array}{ccccc} TTS & \xrightarrow{T\lambda} & TST & \xrightarrow{\lambda T} & STT \\ {\scriptstyle \mu^T S}\downarrow & & & & \downarrow{\scriptstyle S\mu^T} \\ TS & & \xrightarrow[\lambda]{\qquad\qquad} & & ST \end{array}$$

Let (M, f) be an element of $TTS(X)$, and let $f = (f_1, f_2)$. Here, (f_1, f_2) is a function from M to $TS(X)$. Thus, f_1 gives another antichain and f_2 outputs a function from that antichain

to $S(X)$.

$$\begin{aligned}\lambda_X \circ \mu^T S(M,f) &= \lambda_X(\bigcup_{w\in M} \mathrm{Min}(\uparrow w \cap \uparrow f_1(w)), w \mapsto (f_2(w))(w)) \\ &= \downarrow\{(\bigcup_{w\in M} \mathrm{Min}(\uparrow w \cap \uparrow f_1(w)), g) \mid g(w) \in (f_2(w))(w)\} \\ &= \overline{\{(\bigcup_{w\in M} \mathrm{Min}(\uparrow w \cap \uparrow f_1(w)), g) \mid g(w) \in (f_2(w))(w)\}}\end{aligned}$$

$$\begin{aligned}S\mu^T\circ\lambda_X T\circ T\lambda_X(M,f) &= S\mu^T \circ \lambda_X T(M, w \mapsto \downarrow\{(f_1(w), g)|g(w) \in (f_2(w))(w)\}) \\ &= S\mu^T(\downarrow\{(M,(h_1,h_2))|h(w) \in \{(f_1(w),g)|g(w) \in (f_2(w))(w)\}\}) \\ &= \overline{\{(\bigcup_{w\in M} \mathrm{Min}(\uparrow w \cap \uparrow h_1(w)), w \mapsto h_2(w)(w))|h(w) \in \{(f_1(w),g)|g(w) \in f_2(w)(w)\}\}} \\ &= \overline{\{(\bigcup_{w\in M} \mathrm{Min}(\uparrow w \cap \uparrow f_1(w)), g)|g(w) \in (f_2(w))(w)\}}\end{aligned}$$

■

We can also get a distributive law with the lower powerdomain in the opposite direction. Define the natural transformation $\lambda' : ST \to TS$ so that for an object U in $ST(X)$:

$$\lambda'_X(U) = \Big(\bigsqcup_{(M,f)\in U} M, w \mapsto \overline{\bigcup_{(M,f)\in U} f \circ \pi_M(w)}\Big)$$

Proposition 3.5. *For any bounded complete domain, X, $\lambda'_X : ST(X) \to TS(X)$ is monotone.*

Proof. If $U \subseteq V$ for two Scott closed sets in $ST(X)$, we must show that

$$\lambda'_X(U) = \Big(\bigsqcup_{(M,f)\in U} M, w \mapsto \overline{\bigcup_{(M,f)\in U} f \circ \pi_M(w)}\Big) \sqsubseteq \Big(\bigsqcup_{(N,g)\in V} N, z \mapsto \overline{\bigcup_{(N,g)\in V} g \circ \pi_N(z)}\Big) = \lambda'_X(V)$$

$\bigsqcup_{(M,f)\in U} M \sqsubseteq_{EM} \bigsqcup_{(N,g)\in V} N$ since $U \subseteq V$. For the second component, if $w \leq z$, then

$$\begin{aligned}\overline{\bigcup_{(M,f)\in U} f \circ \pi_M(w)} &= \overline{\bigcup_{(M,f)\in U} f \circ \pi_M(z)} \\ &\sqsubseteq \overline{\bigcup_{(N,g)\in V} g \circ \pi_N(z)}\end{aligned}$$

■

Proposition 3.6. *For a bounded complete domain X, $\lambda'_X : ST(X) \to TS(X)$ is Scott continuous.*

Proof. Let $\{U_i\}$ be a directed set in $ST(X)$. Then the supremum is $\overline{\bigcup_i U_i}$.

$$\begin{aligned}\lambda'_X\Big(\overline{\bigcup_i U_i}\Big) &= \Big(\bigsqcup_{(M,f)\in\overline{\bigcup_i U_i}} M,\ w \mapsto \overline{\bigcup_{(M,f)\in\overline{\bigcup_i U_i}} f\circ\pi_M(w)}\Big)\\ &= \Big(\bigsqcup_i \bigsqcup_{(M,f)\in U_i} M,\ w\mapsto \overline{\bigcup_i \bigcup_{(M,f)\in U_i} f\circ\pi_M(w)}\Big)\end{aligned}$$

$$\begin{aligned}\bigsqcup_i \lambda'_X(U_i) &= \bigsqcup_i \Big(\bigsqcup_{(M,f)\in U_i} M, w\mapsto \overline{\bigcup_{(M,f)\in U_i} f\circ\pi_M(w)}\Big)\\ &= \Big(\bigsqcup_i \bigsqcup_{(M,f)\in U_i} M, w\mapsto \overline{\bigcup_i \overline{\bigcup_{(M,f)\in U_i} f\circ\pi_M(w)}}\Big)\\ &= \Big(\bigsqcup_i \bigsqcup_{(M,f)\in U_i} M,\ w\mapsto \overline{\bigcup_i \bigcup_{(M,f)\in U_i} f\circ\pi_M(w)}\Big)\end{aligned}$$

■

Proposition 3.7. *λ' is a natural transformation.*

Proof. Suppose X and Y are bounded complete domains and $a : X \to Y$ is a Scott continuous function. Again, if λ' is a natural transformation, then the following diagram must commute:

$$\begin{array}{ccc} ST(X) & \xrightarrow{ST(a)} & ST(Y) \\ {\scriptstyle \lambda'_X}\downarrow & & \downarrow{\scriptstyle \lambda'_Y} \\ TS(X) & \xrightarrow[TS(a)]{} & TS(Y) \end{array}$$

We check this here:

$$\begin{aligned}\lambda'_Y \circ ST(a)(U) &= \lambda'_Y \overline{\{(M, a\circ f) | (M,f)\in U\}}\\ &= \Big(\bigsqcup_{(M,f)\in U} M, w\mapsto \overline{\bigcup_{(M,f)\in U} a\circ f\circ\pi_M(w)}\Big)\end{aligned}$$

$$
\begin{aligned}
TS(a) \circ \lambda'_X U &= TS(a)\Big(\bigsqcup_{(M,f)\in U} M, w \mapsto \overline{\bigcup_{(M,f)\in U} f \circ \pi_M(w)} \Big) \\
&= \Big(\bigsqcup_{(M,f)\in U} M, w \mapsto \overline{a\big(\overline{\bigcup_{(M,f)\in U} f \circ \pi_M(w)} \big)} \Big) \\
&= \Big(\bigsqcup_{(M,f)\in U} M, w \mapsto \overline{a\big(\bigcup_{(M,f)\in U} f \circ \pi_M(w) \big)} \Big) \\
&= \Big(\bigsqcup_{(M,f)\in U} M, w \mapsto \overline{\bigcup_{(M,f)\in U} a \circ f \circ \pi_M(w)} \Big)
\end{aligned}
$$

■

Proposition 3.8. *There is a distributive law between the monad of random choice and the lower powerdomain, using the natural transformation λ'.*

Proof. $[\boldsymbol{\lambda' \circ S\eta^T = \eta^T S}]$

$$
\begin{array}{ccc}
S & \xrightarrow{S\eta^T} & ST' \\
& {\scriptstyle \eta^T S} \searrow & \downarrow {\scriptstyle \lambda} \\
& & TS
\end{array}
$$

Let U be a Scott closed set of X.

$$
\begin{aligned}
\lambda'_X \circ S\eta^T(U) &= \lambda'_X(\{(\epsilon, \chi_u) \mid u \in U\}) \\
&= (\epsilon, \chi_U)
\end{aligned}
$$

$$
\eta^T S(U) = (\epsilon, \chi_U)
$$

$[\boldsymbol{\lambda' \circ \eta^S T = T\eta^S}]$

$$
\begin{array}{ccc}
T & \xrightarrow{\eta^S T} & ST' \\
& {\scriptstyle T\eta^S} \searrow & \downarrow {\scriptstyle \lambda} \\
& & TS
\end{array}
$$

Let (M, f) be in $T(X)$.

$$\begin{aligned}\lambda'_X \circ \eta^S T(M,f) &= \lambda'_X(\downarrow(M,f)) \\ &= (M, w \mapsto \downarrow f(w))\end{aligned}$$

$$T\eta^S(M,f) = (M, w \mapsto \downarrow f(w))$$

$[\boldsymbol{\lambda' \circ S\mu^T = \mu^T S \circ T\lambda' \circ \lambda' T}]$

$$\begin{array}{ccccc} STT & \xrightarrow{\lambda' T} & TST & \xrightarrow{T\lambda'} & TTS \\ {\scriptstyle S\mu^T}\downarrow & & & & \downarrow{\scriptstyle \mu^T S} \\ ST' & & \xrightarrow[\lambda]{} & & TS \end{array}$$

Let U be a closed set in $T^2(X)$. Each element of U can be represented by $(M_i, (f_{i1}, f_{i2}))$, where $f_{i1} : M_i \to FAC(\{0,1\}^\infty)$ is a function that sends a w in M_i to another antichain, N_i^w, and f_{i2} takes a w in M_i and outputs a function $g : N_i^w \to X$.

$$\begin{aligned}\lambda'_X \circ S\mu^T(U) &= \lambda'_X\Big(\overline{\{(\bigcup_{w\in M_i} \mathrm{Min}(\uparrow w \cap \uparrow f_{i1}(w)), z \mapsto (f_{i2}(z))(z))\}_i}\Big) \\ &= \Big(\bigsqcup_i (\bigcup_{w\in M_i} \mathrm{Min}(\uparrow w \cap \uparrow f_{i1}(w))), z \mapsto \overline{\bigcup_i (f_{i2}(z))(z)}\Big) \qquad (1)\end{aligned}$$

$$\begin{aligned}\mu^T S \circ T\lambda'_X \circ \lambda'_X T(U) &= \mu^T S \circ T\lambda'_X\Big(\bigsqcup_i M_i, w \mapsto \overline{\bigcup_i (f_{i1} \circ \pi_{M_i}(w), f_{i2}(w))}\Big) \\ &= \mu^T S\Big(\bigsqcup_i M_i, w \mapsto \big(\bigsqcup_i f_{i1} \circ \pi_{M_i}(w), z \mapsto \overline{\bigcup_i (f_{i2}(w))(z)}\big)\Big) \\ &= \Big(\bigcup_{w\in \sqcup_i M_i} \mathrm{Min}(\uparrow w \cap \uparrow \bigsqcup_i f_{i1} \circ \pi_{M_i}(w)), z \mapsto \overline{\bigcup_i (f_{i2}(z))(z)}\Big) \\ &= \Big(\bigcup_{w\in \sqcup_i M_i} \bigsqcup_i \mathrm{Min}(\uparrow w \cap \uparrow f_{i1} \circ \pi_{M_i}(w)), z \mapsto \overline{\bigcup_i (f_{i2}(z))(z)}\Big) \qquad (2)\end{aligned}$$

It must be shown that the antichains of (1) and (2) are equal. Clearly, they are both above the antichain $\bigsqcup_i M_i$. Now fix an arbitrary z in $\bigsqcup_i M_i$. We can just show that the portions of

(1) and (2) above z are equal. For (2), this portion is clearly

$$\bigsqcup_i \mathrm{Min}(\uparrow z \cap \uparrow f_{i1} \circ \pi_{M_i}(z))$$

In (1), for each i, there is only one $w \in M_i$ that can contribute to the portion above z. This is the unique w below z, $\pi_{M_i}(z)$. Plugging this in gives the equality.

[$\boldsymbol{\lambda' \circ \mu^S T = T\mu^S \circ \lambda' S \circ S\lambda'}$]

$$\begin{array}{ccccc} SST & \xrightarrow{S\lambda'} & STS & \xrightarrow{\lambda' S} & TSS \\ {\scriptstyle \mu^S T}\downarrow & & & & \downarrow{\scriptstyle T\mu^S} \\ ST' & & \xrightarrow[\lambda]{} & & TS \end{array}$$

Let $\{\{(M_{ij}, f_{ij})\}_j\}$ be a closed set of closed sets in $T(X)$.

$$\begin{aligned} \lambda'_X \circ \mu^S T(\{\{(M_{ij}, f_{ij})\}_j\}) &= \lambda'_X\Big(\overline{\bigcup_{i,j}(M_{ij}, f_{ij})}\Big) \\ &= \Big(\bigsqcup_{i,j} M_{ij}, w \mapsto \overline{\bigcup_{i,j} f_{ij} \circ \pi_{M_{ij}}(w)}\Big) \end{aligned}$$

$$\begin{aligned} T\mu^S \circ \lambda'_X S \circ S\lambda'_X(\{\{(M_{ij}, f_{ij})\}_j\}) &= T\mu^S \circ \lambda'_X S\Big(\overline{\big\{\big(\bigsqcup_i M_{ij}, w \mapsto \overline{\bigcup_i f_{ij} \circ \pi_{M_{ij}}(w)}\big)\big\}_j}\Big) \\ &= T\mu^S\Big(\bigsqcup_j \bigsqcup_i M_{ij}, w \mapsto \overline{\bigcup_j \overline{\bigcup_i f_{ij} \circ \pi_{M_{ij}}(w)}}\Big) \\ &= \Big(\bigsqcup_{i,j} M_{ij}, w \mapsto \overline{\bigcup_{i,j} f_{ij} \circ \pi_{M_{ij}}(w)}\Big) \end{aligned}$$

■

Unfortunately, similar distributive laws do not hold for the upper powerdomain. For the first distributive law with the lower powerdomain, it may seem natural to replace the lower set with the upper set to make it work with the upper powerdomain. In this case, if S is the upper powerdomain functor, then the natural transformation $\lambda : TS \to ST$ would be defined for an object (M, f) in $TS(X)$ by:

$$\lambda_X(M, f) = \uparrow\{(M, g) \mid g(w) \in f(w), \forall w \in M\}$$

This does satisfy the laws for the distributive law, but λ_X is not necessarily monotone. For example,

suppose a and b are two incomparable elements of X. Now consider two elements of $TS(X)$, (ϵ, f) and $(\{0,1\}, g)$, where $f(\epsilon) = \uparrow a \cup \uparrow b$, $g(0) = \uparrow a$, and $g(1) = \uparrow b$. Then, $(\epsilon, f) \sqsubseteq (\{0,1\}, g)$. If λ_X were monotone, then

$$\lambda_X(\{0,1\}, g) = \uparrow\{(\{0,1\}, h) \mid h(0) \in \uparrow a, h(1) \in \uparrow b\} \subseteq \uparrow\{(\epsilon, \chi_x) \mid x \in \uparrow a \cup \uparrow b\} = \lambda_X(\epsilon, f)$$

However, consider $(\{0,1\}, h)$, where $h(0) = a$ and $h(1) = b$. Then $(\{0,1\}, h) \in \lambda_X(\{0,1\}, g)$. But if $(\{0,1\}, h) \in \lambda_X(\epsilon, f)$, then it is above some (ϵ, χ_x). Thus, $x \sqsubseteq a, b$, and since a and b do not compare, $x \notin \uparrow a \cup \uparrow b$. Therefore, λ_X is not monotone.

In the second distributive law with the lower powerdomain, the first component of the natural transformation involves taking the supremum of the antichains from a Scott closed set of random choices. For the upper powerdomain, there is an upper set of random choices, so the supremum of the antichains would always be the maximal antichain, $\{0,1\}^\omega$. Another idea would be to take the infimum of the antichains. However, using this natural transformation, one of the commutative diagrams for the distributive law does not hold. We omit any further details, but in Section 3.5, we show how the monad can be altered to get a distributive law with the upper powerdomain.

3.3 Extending the Monad

As has been shown, the random choice functor is a monad on the category BCD. However, only two of the nondeterministic powerdomains (the upper and lower) leave BCD invariant. BCD is not closed under the convex powerdomain, but the Cartesian closed categories RB and FS, which contain BCD, are. The random choice monad is not believed to stay within these categories, since we see no way to construct the required deflations. In BCD, infima can be used, but outside of BCD, infima are not guaranteed. One way to repair this is to not only define the functions on antichains, but instead to define them on the Scott closure, or lower set, of these antichains. This way, there is no need for infima to project down to smaller trees, since the function is already defined on the lower set.

In our original monad, antichains in the Cantor tree are used, representing the possible outcomes of a random computation. Now we change this monad to include not only antichains of words, but also the prefixes of these words. These prefixes represent intermediate stages of computation where more random bits are still needed.

Definition 3.9. A Scott-closed set M in $\{0,1\}^\infty$ is *full* if $\forall w \in M, w*0 \in M \Leftrightarrow w*1 \in M$. Denote the family of nonempty, full Scott-closed subsets of $\{0,1\}^\infty$ by $\Gamma_f(\{0,1\}^\infty)$.

$\Gamma_f(\{0,1\}^\infty)$, ordered by inclusion, is a subposet of the lower powerdomain of $\{0,1\}^\infty$. If a poset is a dcpo, domain, or bounded complete domain, then so is the lower powerdomain of that poset. The supremum of some nonempty subset $\{M_i\}$ is simply the closure of the union, $\overline{\bigcup_i M_i}$.

3.3.1 Properties of $\Gamma_f(\{0,1\}^\infty)$

Proposition 3.10. *$\Gamma_f(\{0,1\}^\infty)$ is a dcpo.*

Proof. A directed set $\{M_i\}$ in $\Gamma_f(\{0,1\}^\infty)$ is also a directed set in the lower powerdomain, $\Gamma(\{0,1\}^\infty)$. This directed set has a supremum, $\overline{\bigcup_i M_i}$. All that needs to be shown is that this supremum is full. The closure only adds infinite words, and infinite words do not affect whether a closed set is full or not. If $w*0 \in \overline{\bigcup_i M_i}$, then there is some i such that $w*0 \in M_i$. Since M_i is full, then $w*1$ is in M_i, and therefore, $w*1 \in \overline{\bigcup_i M_i}$. If $w*1 \in \overline{\bigcup_i M_i}$ the same argument shows that $w*0$ is as well. ■

Proposition 3.11. *$\Gamma_f(\{0,1\}^\infty)$ is a complete lattice.*

Proof. Again, the supremum of any nonempty subset is the closure of the union. The same argument from Proposition 3.10 proves that this supremum is a full Scott-closed set. The closed set, $\{\epsilon\}$, only containing the empty word, serves as the bottom element. The infimum of any nonempty subset is the intersection. The intersection of closed sets is again closed, and since any closed set contains the empty word, the intersection is nonempty. ■

For any full Scott-closed subset, M, and any natural number n, a projection of M can be made to a Scott-closed set of words with length less than or equal to n. A proposition similar to Theorem 2.19 will be useful.

Proposition 3.12. *Let M be in $\Gamma_f(\{0,1\}^\infty)$. Then $M = \bigsqcup_n \pi_n(M)$, where $\pi_n(M)$ is the subset of all words in M of length less than or equal to n.*

Lemma 3.13. *The finite elements of $\Gamma_f(\{0,1\}^\infty)$ are precisely the finite full Scott closed subsets.*

Proof. A finite subset only has a finite number of elements below it, so it is clearly way below itself. If a subset contains an infinite word, then from the above proposition, it is the supremum of some of its finite subsets, so it cannot be way below itself. ■

Now the following proposition is clear.

Proposition 3.14. *$\Gamma_f(\{0,1\}^\infty)$ is an algebraic domain.*

Proof. From Proposition 3.12, each element M is the supremum of the set containing $\pi_n(M)$ for each natural number n. Each $\pi_n(M)$ is finite, so M is the supremum of finite elements below it. Also, the set of finite elements below some M is clearly directed since the supremum of two finite full Scott closed sets is simply their union, which must also be finite.

■

Corollary 3.15. *$\Gamma_f(\{0,1\}^\infty)$ is a bounded complete domain.*

Proof. From Proposition 3.11, $\Gamma_f(\{0,1\}^\infty)$ is a complete lattice, and from the previous proposition, it is a continuous lattice. Therefore, it is a bounded complete domain. ■

3.3.2 The Extended Functor

Now we define the extended functor, which is the same as the previous one except the antichains are replaced with Scott closed sets.

Definition 3.16. For a category of domains, functor RC' is now defined on objects by

$$RC'(D) = \{(M, f) \mid M \in \Gamma_f(\{0,1\}^\infty), f : M \to D \text{ is Scott continuous}\}$$

For $a : D \to D'$ and $(M, f) \in RC'(D)$, we define

$$RC'(a)(M, f) = (M, a \circ f)$$

$RC'(D)$ is ordered by $(M, f) \sqsubseteq (N, g)$ iff $M \subseteq N$ and $f(w) \leq g(w), \forall w \in M$.

Any element (M, f) in $RC'(D)$ can be extended to a bigger Scott closed set N in a minimal way. This represents a situation when you are given more random bits than you need. Getting the extra bits builds up the binary tree but does not change the second component, the function. Thus, for this portion of the binary tree, the function is constant.

Lemma 3.17. *Suppose (M, f) is in $RC'(D)$. For any N such that $M \subseteq N$, there is a least element $(N, \overline{f})$ such that $(M, f) \sqsubseteq (N, \overline{f})$. The function $\overline{f}$ is defined by $\overline{f}(w) = f \circ \pi_M(w)$.*

Proof. Since $M \subseteq N$, $(M, f) \sqsubseteq (N, \overline{f})$ if $f(w) \sqsubseteq \overline{f}(w)$ for all $w \in M$. To minimize $\overline{f}$, $f(w)$ should equal $\overline{f}(w)$ for w in M. This is true for the given function since $f \circ \pi_M(w) = f(w)$. The function

$\overline{f}$ must defined on all of N, and since it must be monotone, the smallest it can be for a given w is $f \circ \pi_M(w)$. This is clearly Scott continuous if f is. ■

Now it is shown that the extended RC' is still a functor in the category of dcpo's.

Proposition 3.18. *If D is a dcpo, then $RC'(D)$ is also a dcpo.*

Proof. Let $\{(M_i, f_i)\}$ be a directed set in $RC'(D)$. The Scott closed sets are ordered by inclusion, so the first component of the supremum must be $\overline{\bigcup_i M_i}$. The supremum of the directed family must be above each (M_i, f_i), and therefore it must be above $(\overline{\bigcup_i M_i}, \overline{f_i})$ for each i, as defined in the previous lemma. Since now the Scott closed set is fixed, the order is solely determined by the pointwise order of the functions. The Scott continuous functions between two dcpos also form a dcpo, so the functions have a supremum, f, defined by $f(w) = \bigsqcup_i \overline{f_i}(w)$. ■

Any full Scott closed set of $\{0,1\}^\infty$ can be viewed as a subdomain of $\{0,1\}^\infty$. This subdomain is also bounded complete. This fact is useful to show that for a fixed M, the set of all (M, f) in $RC'(D)$ is a bounded complete domain if D is a bounded complete domain. This is since each f is a function from M to D, and BCD is Cartesian closed.

Lemma 3.19. *Any element of $\Gamma_f(\{0,1\}^\infty)$ is itself a bounded complete domain.*

Proof. $\{0,1\}^\infty$ is a bounded complete domain. An element M in $\Gamma_f(\{0,1\}^\infty)$ is Scott closed, so it is a dcpo. As a lower set, for any w in M, $\downarrow w \subseteq M$, so M is a domain. Also, any nonempty subset of M has an infimum which must also be in M, so M is bounded complete. ■

The following lemma is obvious.

Lemma 3.20. *Let M be in $\Gamma_f(\{0,1\}^\infty)$. Now define π_M so that $\pi_M(N, g) = (M \cap N, g)$ where g is restricted to the domain $M \cap N$. Then π_M is Scott continuous and if $(M, f) \sqsubseteq (N, g)$ then $(M, f) \sqsubseteq \pi_M(N, g)$.*

Again, if M is fixed, then the set of all (M, f) in $RC'(D)$ is simply the function space $[M \to D]$.

Lemma 3.21. *Suppose D is in a Cartesian closed category of domains and M is a finite element of $\Gamma_f(\{0,1\}^\infty)$. For continuous functions $f, g : M \to D$, if $f \ll g$ in $[M \to D]$, then $(M, f) \ll (M, g)$.*

Proof. Let $\{(M_i, f_i)\}$ be a directed family such that $(M, g) \sqsubseteq \bigsqcup_i (M_i, f_i)$. Then $\{\pi_M(M_i, f_i)\}$ is directed with a supremum above (M, g). This supremum has M as its first component, and since M

is finite, there must be a cofinal subset of the directed family all with M as its first component. With M fixed, the order is the pointwise order of the functions. Thus, for a directed set $\{(M, g_i)\}$ with supremum above (M, g), there is an element above (M, f). This element is of the form $\pi_M(M_i, f_i)$ for some i, so (M_i, f_i) is also above (M, f). ■

Corollary 3.22. *If $(N, g) \ll (M, f)$, then N is finite and $g \ll f$ where g and f are both viewed as functions in $[N \to D]$.*

In the functor using antichains, it was shown that every element (M, f) is the supremum of all $\pi_n(M, f)$, the projections to antichains with words of length at most n. An analogous result is shown here.

Lemma 3.23. *For (M, f) in $RC'(D)$, $(M, f) = \bigsqcup_n \pi_n(M, f)$, where $\pi_n(M, f) = (\pi_n(M), f)$ and $\pi_n(M)$ is the subset of M only containing words of length at most n.*

Proof.

$$\begin{aligned}
\bigsqcup_n \pi_n(M, f) &= (\bigcup_n \pi_n(M), w \mapsto \bigsqcup_n f \circ \pi_{\pi_n(M)}(w)) \\
&= (M, w \mapsto f(\bigsqcup_n \pi_{\pi_n(M)}(w))) \\
&= (M, w \mapsto f(w)) \\
&= (M, f)
\end{aligned}$$

This is true since f is Scott continuous and every word is the supremum of its finite prefixes. ■

Now we can show that RC' is an endofunctor in our desired categories of domains.

Proposition 3.24. *If D is in BCD, RB, or FS, then $RC'(D)$ is a domain.*

Proof. Each (M, f) is supremum of $\pi_n(M, f)$ for all n. For a fixed n, $\pi_n(M)$ is a bounded complete domain, so $[\pi_n(M) \to D]$ is a domain (since we are in a Cartesian closed category of domains). Thus f restricted to $\pi_n(M)$ is the supremum of all functions way below it. Therefore:

$$(M, f) = \bigsqcup_n \{(\pi_n(M), g_n) \mid g_n \ll f)\}$$

and all such elements are way below (M, f) so $(M, f) = \bigsqcup \downarrow (M, f)$.

Now we must show that $\downarrow(M, f)$ is directed. Consider (N, g) and (L, h), both way below (M, f). Then N and L are finite subsets of M, and g and h are way below f when restricted to their domains. For the first component, we can form $N \cup L$, which is also a finite subset of M. Now extend both g and h in the standard way to get $\overline{g}$ and $\overline{h}$. These are both way below f restricted to $N \cup L$. Both $N \cup L$ and D are domains, so their function space, $[N \cup L \to D]$, must also be a domain (again, since we are in a CCC of domains). Thus, the set of functions way below f is directed, so there is an i way below f and above both $\overline{g}$ and $\overline{h}$. Therefore, $(N \cup L, i)$ is way below (M, f) and is above (N, g) and (L, h), proving that $\downarrow(M, f)$ is directed. ■

Theorem 3.25. *RC' is an endofunctor in the category BCD*

Proof. All that is left to show is that $RC'(D)$ is bounded complete. Let $\{(M_i, f_i)\}$ be a nonempty subset in $RC'(D)$. Then the infimum is $(\bigcap_i M_i, \inf_i f_i)$. Restricting each f_i to $\bigcap_i M_i$ gives a set of functions between two bounded complete domains. Since BCD is Cartesian closed, the function space is also bounded complete, so the set of functions has an infimum. ■

Theorem 3.26. *RC' is an endofunctor in the category RB*

Proof. If D is in RB, there is a directed family of deflations $\{d_i\}$ whose supremum is the identity. For each (M, f) in $RC'(D)$, $(M, f) = \bigsqcup_n \pi_n(M, f)$. Then $RC'(d_i) \circ \pi_n$, where for each (M, f), $\big(RC'(d_i) \circ \pi_n\big)(M, f) = (\pi_n(M), d_i \circ f)$ is a deflation, and the supremum of all such deflations is the identity. Thus, $RC'(D)$ is in RB. ■

Another characterization of the category FS from [25] will be helpful to show that RC' is an endofunctor in the category.

Definition 3.27. Let D be a dcpo and $G \subseteq [D \to D]$ be a set of functions below id_D. Then G is *finitely separating* if given a finite sequence $(x_1, \ldots, x_n) \in D^n$ and corresponding sequence $(y_1, \ldots, y_n)$, where $y_i \ll x_i$, there is an element f of G which satisfies $y_i \sqsubseteq f(x_i) \sqsubseteq x_i$ for all i.

Theorem 3.28. *A domain is in FS if and only if the set of deflations is finitely separating.*

Theorem 3.29. *RC' is an endofunctor in the category FS*

Proof. Suppose D is in FS. Let $\{(M_i, f_i)\}_i$ and $\{(N_i, g_i)\}_i$ be finite sequences in $RC'(D)$ such that $(N_i, g_i) \ll (M_i, f_i)$. Then each N_i is finite, and for any $w \in N_i$, $g_i(w) \ll f_i(w)$. There are finitely many N_i each with finitely many words w. Thus, we can form the finite sequence $\{f_i(w) \mid w \in N_i\}_i$

along with the corresponding sequence $\{g_i(w) \mid w \in N_i\}_i$. Since D is in FS, there is a deflation d such that $g_i(w) \sqsubseteq d \circ f_i(w) \sqsubseteq f_i(w)$ for all i and $w \in N_i$.

Now let $N = \bigcup_i N_i$. Define the function $h : RC'(D) \to RC'(D)$ by:

$$h(M, f) = (M \cap N, d \circ f)$$

This is a deflation since N is finite and d is a deflation. $N_i = N_i \cap N \subseteq M_i \cap N \subseteq M_i$, so $(N_i, g_i) \sqsubseteq h(M_i, f_i) = (M_i \cap N, d \circ f_i) \sqsubseteq (M_i, f_i)$. Therefore, the set of deflations in $RC'(D)$ is finitely separating, and $RC'(D)$ is in FS. ■

3.3.3 The Monad

For the monad construction, the unit is the same as before:

$$\eta(d) = (\epsilon, \chi_d)$$

When restricted to the maximal elements of a Scott closed tree, its leaves, our new Kleisli extension is the same as the previous Kleisli extension. It is defined on each component by:

$$\pi_1 \circ h^\dagger(M, f) = M \cup (\bigcup_{w \in M} (\uparrow w \cap (\pi_1 \circ h \circ f(w))))$$

$$((\pi_2 \circ h^\dagger)(M, f))(z) = g(\pi_N(z))$$

where $(N, g) = h \circ f \circ \pi_M(z)$.

Using the notation introduced in Section 2.3.2, the Kleisli extension can be defined as:

$$\pi_1 \circ h^\dagger(M, f) = M \cup (\bigcup_{w \in M} (\uparrow w \cap (h_1^f(w))))$$

$$((\pi_2 \circ h^\dagger)(M, f))(z) = (h_2^f(z))(z)$$

Proposition 3.30. *$h^\dagger$ is monotone.*

Proof. For a given h, it must be shown that if $(M, f) \sqsubseteq (N, g)$, then $h^\dagger(M, f) \sqsubseteq h^\dagger(N, g)$. If

$(M, f) \sqsubseteq (N, g)$, then $M \subseteq N$ and $f(w) \leq g(w)$ for any $w \in M$, and for the first component,

$$\begin{aligned}
\pi_1 \circ h^\dagger(M, f) &= M \cup (\bigcup_{w \in M} (\uparrow w \cap (\pi_1 \circ h \circ f(w)))) \\
&\subseteq N \cup (\bigcup_{w \in M} (\uparrow w \cap (\pi_1 \circ h \circ f(w)))) \\
&\subseteq N \cup (\bigcup_{w \in N} (\uparrow w \cap (\pi_1 \circ h \circ g(w)))) \\
&= \pi_1 \circ h^\dagger(N, g)
\end{aligned}$$

Now we check the second component. For any word $w \in (\pi_1 \circ h^\dagger(M, f))$, we must show that $(\pi_2 \circ h^\dagger(M, f))(w) \sqsubseteq (\pi_2 \circ h^\dagger(N, g))(w)$.

$$\begin{aligned}
(\pi_2 \circ h^\dagger(M, f))(w) &= (\pi_2 \circ h \circ f \circ \pi_M(w))(\pi_{\pi_1 \circ h \circ f \circ \pi_M(w)}(w)) \\
&\sqsubseteq (\pi_2 \circ h \circ g \circ \pi_N(w))(\pi_{\pi_1 \circ h \circ g \circ \pi_N(w)}(w)) \\
&= (\pi_2 \circ h^\dagger(N, g))(w)
\end{aligned}$$

■

Theorem 3.31. *$h^\dagger$ is Scott continuous.*

Proof. It must be shown that $h^\dagger$ preserves directed suprema. Let $\{(M_i, f_i)\}$ be a directed family

in $RC'(D)$. First, the first component is checked.

$$\begin{aligned}
\bigsqcup_i \pi_1 \circ h^\dagger(M_i, f_i) &= \bigsqcup_i \big(M_i \cup (\bigcup_{w \in M_i} (\uparrow w \cap (\pi_1 \circ h \circ f_i(w))))\big) \\
&= \overline{\bigcup_i M_i \cup (\bigcup_{w \in M_i} (\uparrow w \cap (\pi_1 \circ h \circ f_i(w))))} \\
&= \overline{(\bigcup_i M_i) \cup (\bigcup_i \bigcup_{w \in M_i} (\uparrow w \cap (\pi_1 \circ h \circ f_i(w))))} \\
&= \overline{(\bigcup_i M_i) \cup (\bigcup_{w \in \cup_i M_i} \bigcup_i (\uparrow w \cap (\pi_1 \circ h \circ f_i(w))))} \\
&= \overline{(\bigcup_i M_i) \cup (\bigcup_{w \in \cup_i M_i} (\uparrow w \cap \bigcup_i (\pi_1 \circ h \circ f_i(w))))} \\
&= \overline{(\bigcup_i M_i) \cup (\bigcup_{w \in \cup_i M_i} (\uparrow w \cap (\pi_1 \circ h \circ (\bigsqcup_i f_i)(w))))} \\
&= (\overline{\bigcup_i M_i}) \cup (\bigcup_{w \in \overline{\cup_i M_i}} (\uparrow w \cap (\pi_1 \circ h \circ (\bigsqcup_i f_i)(w)))) \\
&= \pi_1 \circ h^\dagger(\overline{\bigcup_i M_i}, \bigsqcup_i f_i)
\end{aligned}$$

Now we check the second component.

$$\begin{aligned}
\bigsqcup_i (\pi_2 \circ h^\dagger(M_i, f_i))(w) &= \bigsqcup_i (\pi_2 \circ h \circ f_i \circ \pi_{M_i}(w))(\pi_{\pi_1 \circ h \circ f_i \circ \pi_{M_i}}(w)) \\
&= \pi_2 \circ h \circ \bigsqcup_i f_i \circ \pi_{M_i}(w))(\pi_{\pi_1 \circ h \circ f_i \circ \pi_{M_i}}(w)) \\
&= (\pi_2 \circ h^\dagger(\sqcup_i M_i, \sqcup_i f_i))(w)
\end{aligned}$$

■

Theorem 3.32. *This extended construction forms a monad.*

Proof. $[\boldsymbol{h^\dagger \circ \eta = h}]$

$$\begin{aligned}
\pi_1 \circ h^\dagger \circ \eta(d) &= \pi_1 \circ h^\dagger(\epsilon, \chi_d)) \\
&= \epsilon \cup (\uparrow \epsilon \cap (\pi_1 \circ h(d))) \\
&= \pi_1 \circ h(d)
\end{aligned}$$

$$\begin{aligned}(\pi_2 \circ h^\dagger \circ \eta(d))(z) &= (\pi_2 \circ h^\dagger(\epsilon, \chi_d))(z) \\ &= (h_2^{\chi_d}(z))(z) \\ &= (\pi_2 \circ h(d))(z).\end{aligned}$$

$[\eta^\dagger = \mathbf{id}]$

$$\begin{aligned}\pi_1 \circ \eta^\dagger(M, f) &= M \cup (\bigcup_{w \in M} (\uparrow w \cap (\eta_1^f(w)))) \\ &= M \cup (\bigcup_{w \in M} (\uparrow w \cap \epsilon)) \\ &= M \cup \epsilon \\ &= M\end{aligned}$$

since $\eta_1^f(w) = \epsilon$ for each w.

$$\begin{aligned}(\pi_2 \circ \eta^\dagger)(M, f)(z) &= \eta_2^f(z)(z) \\ &= \chi_{f(z)}(z) \\ &= f(z)\end{aligned}$$

$[k^\dagger \circ h^\dagger = (h^\dagger \circ h)^\dagger]$

Let w be any maximal word in M. The Kleisli extension can only make the first component larger. We show that the portion above each w is equal in both cases.

$$\begin{aligned}\uparrow w \cap (\pi_1 \circ k^\dagger \circ h^\dagger(M, f)) &= w \cup (\uparrow w \cap h_1^f(w)) \cup (\bigcup_{z \in w \cup (\uparrow w \cap h_1^f(w))} (\uparrow z \cap k_1^{h_2^f(z)}(z))) \\ &= w \cup (\uparrow w \cap h_1^f(w)) \cup (\uparrow w \cap k_1^{h_2^f(w)}(w)) \cup \bigcup_{z \in \uparrow w \cap h_1^f(w)} (\uparrow z \cap k_1^{h_2^f(z)}(z)) \quad (1)\end{aligned}$$

$$\begin{aligned}
\uparrow w \cap (\pi_1 \circ (k^\dagger \circ h)^\dagger(M,f)) &= w \cup (\uparrow w \cap \pi_1 \circ k^\dagger \circ h \circ f(w)) \\
&= w \cup (\uparrow w \cap (h_1^f(w) \cup (\bigcup_{z \in h_1^f(w)} (\uparrow z \cap k_1^{h_2^f(w)}(z))))) \\
&= w \cup (\uparrow w \cap (h_1^f(w)) \cup (\uparrow w \cap \bigcup_{z \in h_1^f(w)} (\uparrow z \cap k_1^{h_2^f(w)}(z))))) \\
&= w \cup (\uparrow w \cap (h_1^f(w)) \cup (\bigcup_{z \in h_1^f(w)} (\uparrow w \cap \uparrow z \cap k_1^{h_2^f(w)}(z))))) \quad (2)
\end{aligned}$$

Now we must show that (1) is equal to (2), or that

$$(\uparrow w \cap k_1^{h_2^f(w)}(w)) \cup \bigcup_{z \in \uparrow w \cap h_1^f(w)} (\uparrow z \cap k_1^{h_2^f(z)}(z)) = \bigcup_{z \in h_1^f(w)} (\uparrow w \cap \uparrow z \cap k_1^{h_2^f(w)}(z)))$$

First suppose that $\uparrow w \cap h_1^f(w)$ is empty. Then clearly, both sides are equal to $\uparrow w \cap k_1^{h_2^f(w)}(w)$. Now assume the intersection is nonempty. Then $w \in h_1^f(w)$. Thus, for the left hand side, the first intersection is redundant. On the right hand side, the union will only be determined by words z above w. In this case, $\uparrow w \cap \uparrow z = \uparrow z$. Therefore, both sides are equal to

$$\bigcup_{z \in \uparrow w \cap h_1^f(w)} (\uparrow z \cap k_1^{h_2^f(z)}(z))$$

Finally, we have to check the functional component.

$$(\pi_2 \circ k^\dagger \circ h^\dagger(M,f))(z) = (k_2^{h_2^f(z)}(z))(z)$$

$$\begin{aligned}
(\pi_2 \circ (k^\dagger \circ h)^\dagger(M,f))(z) &= ((k^\dagger \circ h)_2^f(z))(z) \\
&= (\pi_2 \circ k^\dagger \circ h \circ f(\pi_M(z)))(z) \\
&= (\pi_2 \circ k^\dagger(h_1^f(w), h_2^f(w)))(z) \\
&= (k_2^{h_2^f(z)}(z))(z)
\end{aligned}$$

■

3.4 Distributive Law With the Convex Powerdomain

Recall that the convex powerdomain of a coherent domain X consists of $Lens(X)$, the nonempty lenses of X. For a nonempty compact $K \subseteq X$, define the *lens closure* of K by $\langle K \rangle = \overline{K} \cap \uparrow K$. The lens closure $\langle K \rangle$ is the smallest lens containing K.

We now show that the random choice monad enjoys a distributive law with the convex powerdomain in the categories RB and FS. Again, we let T be the random choice monad and let S be the convex powerdomain. The unit of the convex powerdomain, $\eta : X \to S(X)$ is defined by:

$$\eta(x) = \{x\}$$

The multiplication, $\mu : S^2(X) \to S(X)$ is defined as:

$$\mu(S) = \langle \bigcup_{U \in S} U \rangle$$

Let a be a morphism from X to Y. For a lens, U, of X, $S(a)(U) = \langle \{a(u) | u \in U\} \rangle$. Objects of $TS(X)$ are random variables (M, f), with $f : M \to Lens(X)$. Objects of $ST(X)$ are lenses of random variables on X. For $(M, f) \in TS(X)$,

$$\begin{aligned} TS(a)(M, f) &= (M, S(a) \circ f) \\ &= (M, w \mapsto \langle \{a(u) | u \in f(w)\} \rangle) \end{aligned}$$

For a lens, U, of $ST(X)$,

$$ST(a)(U) = \left\langle \{(M, a \circ f) \mid (M, f) \in U\} \right\rangle$$

Suppose $U \in ST(X)$. Define the natural transformation, $\lambda : ST \to TS$, so that for an object U in $ST(X)$:

$$\lambda_X(U) = \left(\overline{\bigcup_{(M,f) \in U} M}, w \mapsto \left\langle \bigcup_{(M,f) \in U} f \circ \pi_M(w) \right\rangle \right)$$

A lemma found in [37] is very helpful in proving the distributive law.

Lemma 3.33. *If $a : X \to Y$ is continuous, then for any compact $K \subseteq X$, $\langle a(K) \rangle = \langle a(\langle K \rangle) \rangle$.*

Now we show λ is a natural transformation in the relevant categories.

Proposition 3.34. *For any domain, X, $\lambda_X : ST(X) \to TS(X)$ is monotone.*

Proof. If $U \sqsubseteq_{EM} V$ for two sets in $ST(X)$, we must show that

$$\lambda_X(U) = (\overline{\bigcup_{(M,f)\in U} M}, w \mapsto \Big\langle \bigcup_{(M,f)\in U} f \circ \pi_M(w) \Big\rangle) \sqsubseteq (\overline{\bigcup_{(N,g)\in V} N}, z \mapsto \Big\langle \bigcup_{(N,g)\in V} g \circ \pi_N(z) \Big\rangle) = \lambda_X(V)$$

For any $(M, f) \in U$, there must be some $(N, g) \in V$ above it so that $M \subseteq N$. Therefore, $\overline{\bigcup_{(M,f)\in U} M} \subseteq \overline{\bigcup_{(N,g)\in U} M}$. For the second component, we must show that

$$\Big\langle \bigcup_{(M,f)\in U} f \circ \pi_M(w) \Big\rangle \sqsubseteq_{EM} \Big\langle \bigcup_{(N,g)\in V} g \circ \pi_N(w) \Big\rangle$$

for any w in $\overline{\bigcup_{(M,f)\in U} M}$. First we show that

$$\bigcup_{(M,f)\in U} f \circ \pi_M(w) \subseteq \downarrow\Big(\bigcup_{(N,g)\in V} g \circ \pi_N(w) \Big)$$

Again, for any $(M, f) \in U$ there is some $(N, g) \in V$ above it, so $f \circ \pi_M(w) \sqsubseteq g \circ \pi_N(w)$. Finally we show that

$$\bigcup_{(N,g)\in V} g \circ \pi_N(w) \subseteq \uparrow\Big(\bigcup_{(M,f)\in U} f \circ \pi_M(w) \Big)$$

For any $(N, g) \in V$ there is some (M, f) below it, so $g \circ \pi_N(w) \sqsupseteq f \circ \pi_M(w)$. ■

Proposition 3.35. *For any domain, X, $\lambda_X : ST(X) \to TS(X)$ is Scott continuous.*

Proof. Let $\{U_i\}$ be a directed set in $ST(X)$. Then the supremum is $\langle \bigcup_i U_i \rangle$.

$$\begin{aligned}
\lambda_X\Big(\langle \bigcup_i U_i \rangle\Big) &= \Big(\bigsqcup_{(M,f)\in\langle \bigcup_i U_i\rangle} M,\ w \mapsto \Big\langle \bigcup_{(M,f)\in\langle \bigcup_i U_i\rangle} f \circ \pi_M(w) \Big\rangle \Big) \\
&= \Big(\bigsqcup_i \bigsqcup_{(M,f)\in U_i} M,\ w \mapsto \Big\langle \bigcup_i \bigcup_{(M,f)\in U_i} f \circ \pi_M(w) \Big\rangle \Big)
\end{aligned}$$

$$\begin{aligned}
\bigsqcup_i \lambda_X(U_i) &= \bigsqcup_i \Big(\bigsqcup_{(M,f)\in U_i} M, w \mapsto \Big\langle \bigcup_{(M,f)\in U_i} f \circ \pi_M(w) \Big\rangle \Big) \\
&= \Big(\bigsqcup_i \bigsqcup_{(M,f)\in U_i} M, w \mapsto \Big\langle \bigcup_i \langle \bigcup_{(M,f)\in U_i} f \circ \pi_M(w) \rangle \Big\rangle \Big) \\
&= \Big(\bigsqcup_i \bigsqcup_{(M,f)\in U_i} M,\ w \mapsto \Big\langle \bigcup_i \bigcup_{(M,f)\in U_i} f \circ \pi_M(w) \Big\rangle \Big)
\end{aligned}$$

■

Proposition 3.36. *λ is a natural transformation.*

Proof. Again, if λ is a natural transformation, then the following diagram must commute:

$$\begin{array}{ccc}
ST(X) & \xrightarrow{ST(a)} & ST(Y) \\
{\scriptstyle \lambda_X}\downarrow & & \downarrow{\scriptstyle \lambda_Y} \\
TS(X) & \xrightarrow[TS(a)]{} & TS(Y)
\end{array}$$

We check this here:

$$\begin{aligned}
\lambda_Y \circ ST(a)(U) &= \lambda_Y \Big\langle \{(M, a \circ f) | (M,f) \in U\} \Big\rangle \\
&= \Big(\overline{\bigcup_{(M,f)\in U} M}, w \mapsto \Big\langle \bigcup_{(M,f)\in U} (a \circ f \circ \pi_M(w)) \Big\rangle \Big)
\end{aligned}$$

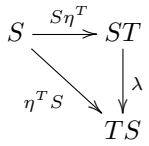
$$\begin{aligned}
TS(a) \circ \lambda_X U &= TS(a) \Big(\overline{\bigcup_{(M,f)\in U} M}, w \mapsto \Big\langle \bigcup_{(M,f)\in U} f \circ \pi_M(w) \Big\rangle \Big) \\
&= \Big(\overline{\bigcup_{(M,f)\in U} M}, w \mapsto \Big\langle a \big(\langle \bigcup_{(M,f)\in U} f \circ \pi_M(w) \rangle \big) \Big\rangle \Big) \\
&= \Big(\overline{\bigcup_{(M,f)\in U} M}, w \mapsto \Big\langle a \big(\bigcup_{(M,f)\in U} f \circ \pi_M(w) \big) \Big\rangle \Big) \\
&= \Big(\overline{\bigcup_{(M,f)\in U} M}, w \mapsto \Big\langle \bigcup_{(M,f)\in U} a \circ f \circ \pi_M(w) \Big\rangle \Big)
\end{aligned}$$

■

Proposition 3.37. *There is a distributive law between the monad of random choice and the convex*

powerdomain, using the natural transformation λ.

Proof. $[\boldsymbol{\lambda \circ S\eta^T = \eta^T S}]$

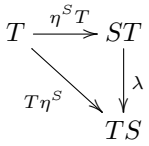


Let U be an element of $S(X)$.

$$\lambda_X \circ S\eta^T(U) = \lambda_X(\{(\epsilon, \chi_u) \mid u \in U\})$$
$$= (\epsilon, \chi_U)$$

$$\eta^T S(U) = (\epsilon, \chi_U)$$

$[\boldsymbol{\lambda \circ \eta^S T = T\eta^S}]$

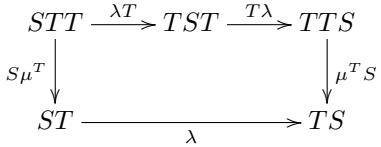


Let (M, f) be an element of $T(X)$.

$$\lambda_X \circ \eta^S T(M, f) = \lambda_X(\{(M, f)\})$$
$$= (M, w \mapsto \{f(w)\})$$

$$T\eta^S(M, f) = (M, w \mapsto \{f(w)\})$$

$[\boldsymbol{\lambda \circ S\mu^T = \mu^T S \circ T\lambda \circ \lambda T}]$



Let $\{(M_i, (f_{i1}, f_{i2}))\}$ be an element of $STT(X)$.

$$\begin{aligned}
\lambda_X \circ S\mu^T(\{(M_i, (f_{i1}, f_{i2}))\}) &= \lambda_X\Big(\Big\langle\{(M_i \cup \bigcup_{w\in M_i} \uparrow w \cap f_{i1}(w), f_{i2} \circ \pi_{M_i})\}\Big\rangle\Big) \\
&= \Big(\overline{\bigcup_i (M_i \cup \bigcup_{w\in M_i} \uparrow w \cap f_{i1}(w))}, z \mapsto \Big\langle\bigcup_i (f_{i2} \circ \pi_{M_i}(z))(z)\Big\rangle\Big) \\
&= \Big(\overline{(\bigcup_i M_i) \cup (\bigcup_i \bigcup_{w\in M_i} \uparrow w \cap f_{i1}(w))}, z \mapsto \Big\langle\bigcup_i (f_{i2} \circ \pi_{M_i}(z))(z)\Big\rangle\Big) \\
&= \Big(\overline{(\bigcup_i M_i) \cup (\bigcup_{w\in\cup_i M_i} \bigcup_i (\uparrow w \cap f_{i1}(w)))}, z \mapsto \Big\langle\bigcup_i (f_{i2} \circ \pi_{M_i}(z))(z)\Big\rangle\Big) \\
&= \Big(\overline{\bigcup_i M_i \cup \bigcup_{w\in\cup_i M_i} (\uparrow w \cap \bigcup_i f_{i1}(w))}, z \mapsto \Big\langle\bigcup_i (f_{i2} \circ \pi_{M_i}(z))(z)\Big\rangle\Big)
\end{aligned}$$

$$\begin{aligned}
\mu^T S \circ T\lambda_X \circ \lambda_X T(\{(M_i, (f_{i1}, f_{i2}))\}) &= \mu^T S \circ T\lambda_X\Big(\overline{\bigcup_i M_i}, w \mapsto \Big\langle\bigcup_i (f_{i1} \circ \pi_{M_i}(w), f_{i2} \circ \pi_{M_i}(w))\Big\rangle\Big) \\
&= \mu^T S\Big(\overline{\bigcup_i M_i}, w \mapsto \Big(\overline{\bigcup_i f_{i1} \circ \pi_{M_i}(w)}, z \mapsto \Big\langle\bigcup_i (f_{i2} \circ \pi_{M_i}(w))(z)\Big\rangle\Big)\Big) \\
&= \Big(\overline{\bigcup_i M_i \cup \bigcup_{w\in\cup_i M_i} (\uparrow w \cap \bigcup_i f_{i1}(w))}, z \mapsto \Big\langle\bigcup_i (f_{i2} \circ \pi_{M_i}(z))(z)\Big\rangle\Big)
\end{aligned}$$

$[\boldsymbol{\lambda \circ \mu^S T = T\mu^S \circ \lambda S \circ S\lambda}]$

$$\begin{array}{ccccc}
SST & \xrightarrow{S\lambda} & STS & \xrightarrow{\lambda S} & TSS \\
{\scriptstyle \mu^S T}\downarrow & & & & \downarrow{\scriptstyle T\mu^S} \\
ST & & \xrightarrow[\lambda]{\qquad\qquad} & & TS
\end{array}$$

Let $\{\{(M_{ij}, f_{ij})\}_j\}$ be an element of $SST(X)$.

$$\begin{aligned}
\lambda_X \circ \mu^S T(\{\{(M_{ij}, f_{ij})\}_j\}) &= \lambda_X\Big(\Big\langle\bigcup_{i,j} (M_{ij}, f_{ij})\Big\rangle\Big) \\
&= \Big(\overline{\bigcup_{i,j} M_{ij}}, w \mapsto \Big\langle\bigcup_{i,j} f_{ij} \circ \pi_{M_{ij}}(w)\Big\rangle\Big)
\end{aligned}$$

$$T\mu^S \circ \lambda_X S \circ S\lambda_X(\{\{(M_{ij}, f_{ij})\}_j\}) = T\mu^S \circ \lambda_X S\Big(\Big\langle \{(\overline{\bigcup_i M_{ij}}, w \mapsto \Big\langle \bigcup_i f_{ij} \circ \pi_{M_{ij}}(w)\Big\rangle)\}_j \Big\rangle\Big)$$
$$= T\mu^S\Big(\overline{\bigcup_j \overline{\bigcup_i M_{ij}}}, w \mapsto \Big\langle \bigcup_j \Big\langle \bigcup_i f_{ij} \circ \pi_{M_{ij}}(w)\Big\rangle\Big\rangle\Big)$$
$$= \Big(\overline{\bigcup_{i,j} M_{ij}}, w \mapsto \Big\langle \bigcup_{i,j} f_{ij} \circ \pi_{M_{ij}}(w)\Big\rangle\Big)$$

■

3.5 Another Variation of the Monad

For the original random choice monad, it was shown that there is a distributive law with the lower powerdomain in both directions. However, there was no distributive law with the upper powerdomain. The problem lies in the first component of the random choice functor, the antichains. We can get around this issue by modifying the order so that the antichains do not factor in. In the original order, $(M, f) \sqsubseteq (N, g)$ if $M \sqsubseteq_{EM} N$ and for all $w \in M$ and $z \in N$ with $w \leq z$, $f(w) \sqsubseteq g(z)$. Now consider a new order such that $(M, f) \sqsubseteq' (N, g)$ if for all $w \in M$ and $z \in N$ such that w and z are comparable, $f(w) \sqsubseteq g(w)$. This is clearly not a partial order, but it is a preorder. We can now form a partial order by making equivalence classes, where $(M, f) \equiv (N, g)$ if $f(w) = g(z)$ for all comparable $w \in M$ and $z \in N$. Every element (M, f) can be extended to $(\{0,1\}^\omega, \overline{f})$, where $\overline{f}(w) = f \circ \pi_M(w)$. Therefore, every equivalence class has an element with $\{0,1\}^\omega$ as the first component. Because of this, the new partial order is the same as just restricting RC to the elements whose antichain is $\{0,1\}^\omega$. With a fixed first component, the elements are merely functions $f : \{0,1\}^\omega \to D$, where f is continuous in the relative Scott topology. Computationally, this represents getting the results of all random coin flips at once, instead of one by one.

3.5.1 The Functor

Definition 3.38. For a category of domains, the functor RC'' is now defined on objects as

$$RC''(D) = [\{0,1\}^\omega \to D]$$

where each function is continuous in the relative Scott topology. The order is simply the pointwise order. For a morphism, $a : D \to E$,

$$RC''(a)(f) = a \circ f$$

For the initial RC'' functor, the supremum is taking componentwise: $\bigsqcup_i (M_i, f_i) = (\bigsqcup_i M_i, \bigsqcup_i \overline{f_i})$.

Thus, the supremum for this new functor coincides with the original. Now, as a subdomain of the first random choice functor, it is clear that given a dcpo, domain, or bounded complete domain, this functor outputs a dcpo, domain, or bounded complete domain, respectively. Finite elements have a member with a finite antichain in its equivalence class. Alternatively, for a finite f, there must be an finite open cover of $\{0,1\}^\omega$ such that f is constant on each open set. Every element is the supremum of functions that are eventually constant in this manner.

3.5.2 The Monad

The unit of the monad is simply the constant function:

$$\eta(d) = \chi_d$$

For a morphism $h : D \to RC''(E)$ the Kleisli extension is defined as:

$$h^\dagger(f)(w) = (h \circ f(w))(w)$$

Therefore, the multiplication of the monad is:

$$\mu(f)(w) = (f(w))(w)$$

Theorem 3.39. *The functor RC'' forms a monad.*

Proof. Now the three monad laws are shown to hold.

$[h^\dagger \circ \eta = h]$

$$\begin{aligned} h^\dagger \circ \eta(d) &= h^\dagger(\chi_d) \\ &= w \mapsto (h \circ \chi_d(w))(w) \\ &= w \mapsto (h \circ d)(w) \\ &= h(d) \end{aligned}$$

$[\boldsymbol{\eta^\dagger = \mathrm{id}}]$

$$\begin{aligned}\eta^\dagger(f) &= w \mapsto (\eta \circ f(w))(w)\\ &= w \mapsto \chi_{f(w)}(w)\\ &= w \mapsto f(w)\\ &= f\end{aligned}$$

$[\boldsymbol{k^\dagger \circ h^\dagger = (h^\dagger \circ h)^\dagger}]$

$$\begin{aligned}k^\dagger \circ h^\dagger(f) &= k^\dagger(w \mapsto (h \circ f(w))(w))\\ &= z \mapsto (k \circ (h \circ f(z))(z))(z)\end{aligned}$$

$$\begin{aligned}(k^\dagger \circ h)^\dagger(f) &= z \mapsto (k^\dagger \circ h \circ f(z))(z)\\ &= z \mapsto (w \mapsto (k \circ (h \circ f(z))(w))(w))(z)\\ &= z \mapsto (k \circ (h \circ f(z))(z))(z)\end{aligned}$$

■

3.6 Distributive Law With the Upper Powerdomain

Now let S be the upper powerdomain functor and let T be this newly defined random choice functor. Define the natural transformation $\lambda : ST \to TS$ so that for a U in $ST(X)$:

$$\lambda_X(U) = w \mapsto \uparrow\{u(w) \mid u \in U\}$$

Proposition 3.40. *For any bounded complete domain, X, $\lambda_X : ST(X) \to TS(X)$ is monotone.*

Proof. If $U \supseteq V$, then

$$\begin{aligned}\lambda_X(U) &= w \mapsto \uparrow\{u(w) \mid u \in U\} \\ &\sqsubseteq w \mapsto \uparrow\{v(w) \mid v \in V\}) \\ &= \lambda_X(V)\end{aligned}$$

■

Proposition 3.41. *For any bounded complete domain, X, $\lambda_X : ST(X) \to TS(X)$ is Scott continuous.*

Proof. Let $\{U_i\}$ be a directed family in $ST(X)$ with supremum $\bigcap_i U_i$. Then

$$\begin{aligned}\bigsqcup_i \lambda_X(U_i) &= \bigsqcup_i (w \mapsto \uparrow\{u(w) \mid u \in U_i\}) \\ &= w \mapsto \bigcap_i \uparrow\{u(w) \mid u \in U_i\} \\ &= w \mapsto \uparrow\bigcap_i \{u(w) \mid u \in U_i\} \\ &= w \mapsto \uparrow\{u(w) \mid u \in \bigcap_i U_i\} \\ &= \lambda_X(\bigcap_i U_i)\end{aligned}$$

■

Proposition 3.42. *λ is a natural transformation.*

Proof. For any morphism $a : X \to Y$, the following diagram must commute:

$$\begin{array}{ccc} ST(X) & \xrightarrow{ST(a)} & ST(Y) \\ {\scriptstyle \lambda_X}\downarrow & & \downarrow{\scriptstyle \lambda_Y} \\ TS(X) & \xrightarrow[TS(a)]{} & TS(Y) \end{array}$$

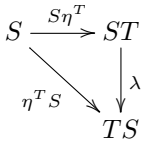
$$\begin{aligned}\lambda_Y \circ ST(a)(U) &= \lambda_Y(\uparrow\{a \circ u \mid u \in U\}) \\ &= w \mapsto \uparrow\{f(w) \mid f \in \uparrow\{a \circ u \mid u \in U\}\} \\ &= w \mapsto \uparrow\{a \circ u(w) \mid u \in U\}\end{aligned}$$

$$\begin{aligned} TS(a) \circ \lambda_X(U) &= TS(a)(w \mapsto \uparrow\{u(w) | u \in U\}) \\ &= w \mapsto \uparrow\{a(x) \mid x \in \uparrow\{u(w) \mid u \in U\}\} \\ &= w \mapsto \uparrow\{a \circ u(w) \mid u \in U\} \end{aligned}$$

■

Proposition 3.43. *In the category of bounded complete domains, there is a distributive law with the upper powerdomain, using the natural transformation* λ.

Proof. $[\boldsymbol{\lambda \circ S\eta^T = \eta^T S}]$

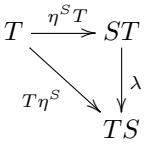


Let U be in $S(X)$.

$$\begin{aligned} \lambda_X \circ S\eta^T(U) &= \lambda_X(\uparrow\{\chi_u \mid u \in U\}) \\ &= w \mapsto \uparrow\{f(w) \mid f \in \uparrow\{\chi_u \mid u \in U\}\} \\ &= w \mapsto \uparrow\{\chi_u(w) \mid u \in U\} \\ &= w \mapsto \uparrow\{u \mid u \in U\} \\ &= w \mapsto U \\ &= \chi_U \end{aligned}$$

$$\eta^T S(U) = \chi_U$$

$[\boldsymbol{\lambda \circ \eta^S T = T\eta^S}]$



Let f be in $T(X)$.

$$\begin{aligned}\lambda_X \circ \eta^S T(f) &= \lambda_X(\uparrow f) \\ &= w \mapsto \uparrow\{g(w) \mid g \in \uparrow f\} \\ &= w \mapsto \uparrow f(w)\end{aligned}$$

$$T\eta^S(f) = w \mapsto \uparrow f(w)$$

$[\boldsymbol{\lambda \circ S\mu^T = \mu^T S \circ T\lambda \circ \lambda T}]$

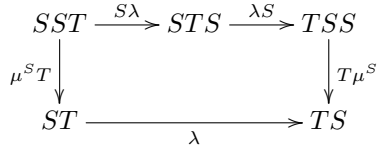
$$\begin{array}{ccccc} STT & \xrightarrow{\lambda T} & TST & \xrightarrow{T\lambda} & TTS \\ {\scriptstyle S\mu^T}\downarrow & & & & \downarrow{\scriptstyle \mu^T S} \\ ST & & \xrightarrow[\lambda]{\qquad\qquad} & & TS \end{array}$$

Let U be in $STT(X)$.

$$\begin{aligned}\mu^T S \circ T\lambda_X \circ \lambda_X T(U) &= \mu^T S \circ T\lambda_X(w \mapsto \uparrow\{u(w) \mid u \in U\}) \\ &= \mu^T S(w \mapsto (z \mapsto \uparrow\{f(z) \mid f \in \uparrow\{u(w) \mid u \in U\}\})) \\ &= \mu^T S(w \mapsto (z \mapsto \uparrow\{(u(w))(z) \mid u \in U\})) \\ &= w \mapsto (z \mapsto \uparrow\{(u(w))(z) \mid u \in U\})(w) \\ &= w \mapsto \uparrow\{(u(w))(w) \mid u \in U\}\end{aligned}$$

$$\begin{aligned}\lambda_X \circ S\mu^T(U) &= \lambda_X(\uparrow\{z \mapsto (u(z))(z) \mid u \in U\}) \\ &= w \mapsto \uparrow\{f(w) \mid f \in \uparrow\{z \mapsto (u(z))(z) \mid u \in U\}\} \\ &= w \mapsto \uparrow\{(z \mapsto (u(z))(z))(w) \mid u \in U\} \\ &= w \mapsto \uparrow\{(u(w))(w) \mid u \in U\}\end{aligned}$$

$[\boldsymbol{\lambda \circ \mu^S T = T\mu^S \circ \lambda S \circ S\lambda}]$



Let U be in $SST(X)$.

$$\begin{aligned}
T\mu^S \circ \lambda_X S \circ S\lambda_X(U) &= T\mu^S \circ \lambda_X S(\uparrow\{z \mapsto \uparrow\{f(z) \mid f \in u\} \mid u \in U\}) \\
&= T\mu^S(w \mapsto \uparrow\{g(w) \mid g \in \uparrow\{z \mapsto \uparrow\{f(z) \mid f \in u\} \mid u \in U\}\}) \\
&= T\mu^S(w \mapsto \uparrow\{(z \mapsto \uparrow\{f(z) \mid f \in u\})(w) \mid u \in U\}) \\
&= T\mu^S(w \mapsto \uparrow\{\uparrow\{f(w) \mid f \in u\} \mid u \in U\}) \\
&= w \mapsto \uparrow \bigcup_{u \in U} \{f(w) \mid f \in u\}
\end{aligned}$$

$$\begin{aligned}
\lambda_X \circ \mu^S T(U) &= \lambda_X(\bigcup_{u \in U} u) \\
&= w \mapsto \uparrow\{f(w) \mid f \in \bigcup_{u \in U} u\} \\
&= w \mapsto \uparrow \bigcup_{u \in U} \{f(w) \mid f \in u\}
\end{aligned}$$

■

Chapter 4

Randomized PCF

4.1 PCF

PCF (Programming Computable Functions) comes from a formal system introduced by Dana Scott in 1969 in a famous paper that remained unpublished until 1993 [38]. PCF augments the simply typed lambda calculus with general recursion and basic arithmetic.

The semantics of PCF presented here come from Gunter's *Semantics of Programming Languages* [39]. They are presented here so they can be directly compared to rPCF, which augments PCF with random choice. The theorems with PCF given below are also presented and proved in [39].

4.1.1 Typing Rules

PCF has just two base types: *nat* and *bool*, with natural numbers and booleans. For a constant natural number n, we use the notation $\underline{n}$ to avoid confusing it with a variable. The two boolean constants are denoted tt and ff. For any two types, there is a function type between them. There are functions, pred and succ, that send a *nat* to a *nat* and a function zero? that sends a *nat* to a *bool*. Then there are basic conditional and recursive statements. Finally, given a function of type $s \to t$, it can applied to something of type s to get something of type t. The typing rules are displayed in Table 4.1.

As explained in Section 1.2.1, H is a type assignment, which is a list of variables and their types. For example, the typing rule [Pred] means that if under type assignment H, M has type *nat*, then pred (M) also has type *nat*.

4.1.2 Equational Rules

Table 4.2 lists basic equations that should hold inside of PCF. Again, for each equation, H represents an arbitrary type assignment. The statement $(H \triangleright M = N : t)$ means that under the type assignment H, M and N are equal terms of type t.

These base equations can be used to prove other equations. For example, suppose we want to prove that pred (succ $(\underline{n})$) = $\underline{n}$. From [Succ], succ $(\underline{n})$ = $\underline{n+1}$. Using this and [Cong], pred (succ $(\underline{n})$) = pred $(\underline{n+1})$. Finally, from [PredSucc], pred $(\underline{n+1}) = \underline{n}$.

4.1.3 Small Step Semantics

Although the equational rules say what should be equal in PCF, they do not say how these equations can be arrived. For these, we can use operational semantics. There are two types of operational semantics: small step semantics and big step semantics. Small step semantics show how

[Nat]	$H \vdash \underline{n} : nat$
[True]	$H \vdash \text{tt} : bool$
[False]	$H \vdash \text{ff} : bool$
[Pred]	$\dfrac{H \vdash M : nat}{H \vdash \text{pred } (M) : nat}$
[Succ]	$\dfrac{H \vdash M : nat}{H \vdash \text{succ } (M) : nat}$
[IsZero]	$\dfrac{H \vdash M : nat}{H \vdash \text{zero? } (M) : bool}$
[If]	$\dfrac{H \vdash L : bool \quad H \vdash M : t \quad H \vdash N : t}{H \vdash \text{if } L \text{ then } M \text{ else } N : t}$
[Lambda]	$\dfrac{H, x : s \vdash M : t}{H \vdash \lambda x : s.M : s \to t)}$
[Rec]	$\dfrac{H, x : t \vdash M : t}{H \vdash \mu x : t.M : t}$
[Ev]	$\dfrac{H \vdash M : s \to t \quad H \vdash N : s}{H \vdash M(N) : t}$

Table 4.1: Typing Rules For PCF

the evaluation of terms occurs step by step. Evaluation occurs one step at a time until a semantic value is reached. For PCF, a semantic value is a natural number, a boolean, or a λ function.

The transition rules for the small step semantics of PCF are shown in Table 4.3. As an example, for a term $M(N)$, the small step semantics say that M should be evaluated first. If $M \to M'$, then $M(N) \to M'(N)$. So if $M \to \lambda x.M'$, then $M(N) \to (\lambda x.M')(N) \to M'[N/x]$. If N can be reached from M in zero or more steps, this is denoted $M \to^* N$.

The small step semantics should preserve the equational rules given above.

Theorem 4.1. *If $H \vdash M : t$ and $M \to N$, then $H \vdash N : t$ and $\vdash (H \triangleright M = N : t)$.*

4.1.4 Big Step Semantics

In big step semantics, a term is evaluated to a value in one step. Thus, only values should appear on the right side of the ($\Downarrow$) operator. The transition rules for PCF's big step semantics are shown in Table 4.4.

The big step semantics should match the transition rules for the small step semantics.

Theorem 4.2. *$M \Downarrow V$ if and only if $M \to^* V$.*

[PredZero] $\vdash (H \triangleright \text{pred } (0) = 0 : nat)$

[PredSucc] $\vdash (H \triangleright \text{pred } (\underline{n+1}) = \underline{n} : nat)$

[ZeroZero] $\vdash (H \triangleright \text{zero? } (0) = \text{tt} \ : bool)$

[ZeroSucc] $\vdash (H \triangleright \text{zero? } (\underline{n+1}) = \text{ff} \ : bool)$

[Succ] $\vdash (H \triangleright \text{succ } (\underline{n}) = \underline{n+1} : nat)$

[Mu] $$\frac{H, x : t \vdash M : t}{\vdash (H \triangleright \mu x : t.M = M[\mu x : t.M/x] : t)}$$

[Lambda] $$\frac{\vdash (H, x : s \triangleright M = N : t}{\vdash (H \triangleright \lambda x : s.M = \lambda x : s.N : s \to t)}$$

[Beta] $$\frac{H, x : s \vdash M : t \qquad H \vdash N : s}{\vdash (H \triangleright (\lambda x : s.M)(N) = M[N/x] : t)}$$

[Eta] $$\frac{H \vdash M : s \to t \qquad x \notin Fv(M)}{\vdash (H \triangleright \lambda x : s.M(x) = M : s \to t)}$$

[IfTrue] $$\frac{H \vdash M : t \qquad H \vdash N : t}{\vdash (H \triangleright \text{if tt then } M \text{ else } N = M : t)}$$

[IfFalse] $$\frac{H \vdash M : t \qquad H \vdash N : t}{\vdash (H \triangleright \text{if ff then } M \text{ else } N = N : t)}$$

[App] $$\frac{\vdash (H \triangleright M = M' : s \to t) \qquad \vdash (H \triangleright N = N' : s)}{\vdash (H \triangleright M(N) = M'(N') : t)}$$

[Cong] $$\frac{\vdash (H \triangleright M = N : nat)}{\vdash (H \triangleright \text{pred } (M) = \text{pred } (N) : nat)}$$

$$\frac{\vdash (H \triangleright M = N : nat)}{\vdash (H \triangleright \text{succ } (M) = \text{succ } (N) : nat)}$$

$$\frac{\vdash (H \triangleright M = N : nat)}{\vdash (H \triangleright \text{zero? } (M) = \text{zero? } (N) : bool)}$$

$$\frac{\vdash (H \triangleright M = N : t)}{\vdash (H \triangleright \mu x : t.M = \mu x : t.N : t)}$$

$$\frac{\vdash (H \triangleright L = L' : bool \qquad \vdash (H \triangleright M = M' : t \qquad \vdash (H \triangleright N = N' : t}{\vdash (H \triangleright \text{if } L \text{ then } M \text{ else } N = \text{if } L' \text{ then } M' \text{ else } N' : t}$$

Table 4.2: Equational Rules For PCF

$$\frac{}{\text{pred } (0) \to 0} \qquad \frac{}{\text{pred } (\text{succ } (\underline{n})) \to \underline{n}}$$

$$\frac{}{\text{zero? } (0) \to \text{tt}} \qquad \frac{}{\text{zero? } (\text{succ } (\underline{n})) \to \text{ff}}$$

$$\frac{}{\text{if tt then } M \text{ else } N \to M}$$

$$\frac{}{\text{if ff then } M \text{ else } N \to N}$$

$$\frac{}{(\lambda x.M)(N) \to M[N/x]}$$

$$\frac{}{\mu x.M \to M[\mu x.M/x]}$$

$$\frac{M \to M'}{\text{pred } (M) \to \text{pred } (M')} \qquad \frac{M \to M'}{\text{zero? } (M) \to \text{zero? } (M')}$$

$$\frac{M \to M'}{\text{succ } (M) \to \text{succ } (M')} \qquad \frac{M \to M'}{M(N) \to M'(N)}$$

$$\frac{L \to L'}{\text{if } L \text{ then } M \text{ else } N \to \text{if } L' \text{ then } M \text{ else } N}$$

Table 4.3: Small Step Semantics For PCF

$$\frac{}{0 \Downarrow 0} \qquad \frac{}{\lambda x.M \Downarrow \lambda x.M}$$

$$\frac{}{\text{tt} \Downarrow \text{tt}} \qquad \frac{}{\text{ff} \Downarrow \text{ff}}$$

$$\frac{M \Downarrow 0}{\text{pred } (M) \Downarrow 0} \qquad \frac{M \Downarrow \underline{n+1}}{\text{pred } (M) \Downarrow \underline{n}}$$

$$\frac{M \Downarrow \underline{n}}{\text{succ } (M) \Downarrow \underline{n+1}} \qquad \frac{M[\mu x.M/x] \Downarrow V}{\mu x.M \Downarrow V}$$

$$\frac{M \Downarrow 0}{\text{zero? } (M) \Downarrow \text{tt}} \qquad \frac{M \Downarrow \underline{n+1}}{\text{zero? } (M) \Downarrow \text{ff}}$$

$$\frac{M \Downarrow \lambda x.M' \qquad N \Downarrow N' \qquad M'[N'/x] \Downarrow V}{M(N) \Downarrow V}$$

$$\frac{B \Downarrow \text{tt} \qquad M \Downarrow V}{\text{if } B \text{ then } M \text{ else } N \Downarrow V}$$

$$\frac{B \Downarrow \text{ff} \qquad N \Downarrow V}{\text{if } B \text{ then } M \text{ else } N \Downarrow V}$$

Table 4.4: Big Step Semantics For PCF

$$[\![H \triangleright x : t]\!]\rho = \rho(x)$$
$$[\![H \triangleright \lambda x : u.M : u \to v]\!]\rho = (d \mapsto [\![H, x : u \triangleright M : v]\!](\rho[x \mapsto d]))$$
$$[\![H \triangleright M(N) : t]\!]\rho = \mathtt{ev}([\![H \triangleright M : s \to t]\!]\rho, [\![H \triangleright N : s]\!]\rho)$$
$$[\![H \triangleright \mu x : t.M : t]\!]\rho = \mathtt{fix}(d \mapsto [\![H, x : t \triangleright M : t]\!]\rho[x \mapsto d])$$
$$[\![\underline{n}]\!]\rho = n$$
$$[\![\mathrm{tt}]\!]\rho = \mathtt{true}$$
$$[\![\mathrm{ff}]\!]\rho = \mathtt{false}$$
$$[\![\text{succ } (M)]\!]\rho = \begin{cases} [\![M]\!]\rho + 1 & [\![M]\!]\rho \neq \bot \\ \bot & [\![M]\!]\rho = \bot \end{cases}$$
$$[\![\text{pred } (M)]\!]\rho = \begin{cases} [\![M]\!]\rho - 1 & [\![M]\!]\rho \neq \bot \\ \bot & [\![M]\!]\rho = \bot \end{cases}$$
$$[\![\text{zero? } (M)]\!]\rho = \begin{cases} \mathtt{true} & [\![M]\!]\rho = 0 \\ \mathtt{false} & [\![M]\!]\rho \neq \bot, 0 \\ \bot & [\![M]\!]\rho = \bot \end{cases}$$
$$[\![H \triangleright \text{if } L \text{ then } M \text{ else } N]\!]\rho = \begin{cases} [\![M]\!]\rho & [\![L]\!]\rho = \mathtt{true} \\ [\![N]\!]\rho & [\![L]\!]\rho = \mathtt{false} \\ \bot & [\![L]\!]\rho = \bot \end{cases}$$

Table 4.5: Denotational Semantics For PCF

4.1.5 Denotational Semantics

Scott's model is a domain theoretic denotational semantics for PCF. For the base term *nat*, let $[\![nat]\!] = \mathbb{N}_\bot$, the flat natural numbers with an added bottom, and $[\![bool]\!] = \mathbb{B}_\bot$, the flat booleans with a bottom. For a type $s \to t$, $[\![s \to t]\!] = [[\![s]\!] \to [\![t]\!]]$, the Scott continuous functions from $[\![s]\!]$ to $[\![t]\!]$. The denotational semantics are given in Table 4.5.

As stated above, each H is a type assignment. For any $x \in H$, $H(x)$ is a type. An *H-environment* is a function ρ that maps each $x \in H$ to a value in $[\![H(x)]\!]$. The function $\rho[x \mapsto d]$ is defined by

$$\rho[x \mapsto d](y) = \begin{cases} d & \text{if } y \equiv x \\ \rho(y) & \text{otherwise} \end{cases}$$

This updates the environment ρ by sending x to d.

Of course, the denotational semantics should relate to the previous equational and transition rules.

Theorem 4.3. (Soundness) *If* $\vdash (H \triangleright M = N : t)$*, then* $[\![H \triangleright M : t]\!] = [\![H \triangleright N : t]\!]$.

Theorem 4.4. (Adequacy) *If M is a closed term of ground type and* $[\![M]\!] = [\![V]\!]$ *for a value V, then* $M \Downarrow V$.

4.1.6 Full Abstraction

Computational adequacy shows that the operational semantics and denotational semantics match for ground types. For a correspondence for arbitrary terms, a notion known as *full abstraction* is used. Before full abstraction can be defined, a relation of operational equivalence must be explained.

A *context* is a term tree with a hole (a missing subterm). This hole can be filled with any given term of the correct type. For a context L with hole x, denote replacing x with a term M by $L\{M/x\}$. Two terms, M and N, are *operationally equivalent* ($M \approx N$), if for any context L, $L\{M/x\} \Downarrow V$ if, and only if, $L\{N/x\} \Downarrow V$. Now, the denotational semantics is said to be fully abstract if for any terms M and N, $[\![M]\!] = [\![N]\!] \Leftrightarrow M \approx N$. The ($\Rightarrow$) direction can be proven from computational adequacy, but the ($\Leftarrow$) direction needs further proof. In fact, Scott's denotational model for PCF is not fully abstract.

The problem is that the denotational semantics has too many elements. Thus, there are operationally indistinguishable programs that have different denotational meanings. For example, consider a *parallel-or* function $por : \mathbb{B}_\perp \to (\mathbb{B}_\perp \to \mathbb{B}_\perp)$ defined by:

$$por(\mathrm{tt})(\perp) = \mathrm{tt}$$

$$por(\perp)(\mathrm{tt}) = \mathrm{tt}$$

$$por(\mathrm{ff})(\mathrm{ff}) = \mathrm{ff}$$

This fully defines the function, by monotonicity. It turns out that there are no programs in PCF whose denotational meaning is *por*. PCF can be extended with such a function so that Scott's model is fully abstract.

In 2000, a syntax-independent, fully abstract model of PCF was found independently by Abramsky, Jagadeesan, and Malacaria [40] and by Hyland and Ong [41], both using game semantics. Semantic models using domains have yet to be shown to be fully abstract.

4.2 Randomized PCF

Now we modify PCF to create Randomized PCF (rPCF), using the same ideas from the random choice monad. We begin by creating a new datatype of binary trees. For any type a', the inductive type rc a' is defined by:

$$\mathrm{rc}\ a' = \mathrm{Leaf}\ a' \mid \mathrm{Node}\ (\mathrm{rc}\ a', \mathrm{rc}\ a')$$

[Zero]	$H \vdash 0 : \text{Leaf } nat$
[True]	$H \vdash \text{tt} : \text{Leaf } bool$
[False]	$H \vdash \text{ff} : \text{Leaf } bool$
[Pred]	$\dfrac{H \vdash M : \text{rc } nat}{H \vdash \text{pred } (M) : \text{rc } nat}$
[Succ]	$\dfrac{H \vdash M : \text{rc } nat}{H \vdash \text{succ } (M) : \text{rc } nat}$
[IsZero]	$\dfrac{H \vdash M : \text{rc } nat}{H \vdash \text{zero? } (M) : \text{rc } bool}$
[If]	$\dfrac{H \vdash L : \text{rc } bool \quad H \vdash M : \text{rc } t \quad H \vdash N : \text{rc } t}{H \vdash \text{if } L \text{ then } M \text{ else } N : \text{rc } t}$
[Lambda]	$\dfrac{H, x : s \vdash M : \text{rc } t}{H \vdash \lambda x : s.M : \text{Leaf } (s \to \text{rc } t)}$
[Rec]	$\dfrac{H, x : t \vdash M : \text{rc } t}{H \vdash \mu x : t.M : \text{rc } t}$
[Ev]	$\dfrac{H \vdash M : \text{rc } (s \to \text{rc } t) \quad H \vdash N : \text{rc } s}{H \vdash M(N) : \text{rc } t}$
[Node]	$\dfrac{H \vdash L : \text{rc } t \quad H \vdash R : \text{rc } t}{H \vdash \text{Node } (L, R) : \text{rc } t}$

Table 4.6: Typing Rules For rPCF

In rPCF, a distinction has to be made between deterministic types and randomized types. The deterministic types are of the form:

$$t ::= nat \mid bool \mid t \to \text{rc } t$$

All types of rPCF are randomized types of the form rc s for some $s \in t$. The terms of rPCF are given by the following grammar:

$$\begin{array}{lll} M, N, L & \texttt{::=} & \underline{n} \texttt{ | tt | ff |} \ x \texttt{ |} \\ & & \text{succ } (M) \texttt{ |} \ \text{pred } (M) \texttt{ |} \ \text{zero? } (M) \texttt{ |} \ \text{if } L \text{ then } M \text{ else } N \texttt{ |} \\ & & \lambda x : t.M \texttt{ |} \ MN \texttt{ |} \ \mu x : t.M \texttt{ |} \ \text{Node } (M, N) \end{array}$$

Note that the terms of rPCF are the same as PCF with the addition of Node (M, M). Note that terms without nodes, such as $\underline{n}$, tt, and $\lambda x : t.M$, are understood to be leaves, but we do not specify it with notation.

4.2.1 Typing Rules

The typing rules for rPCF are given in Table 4.6. These rules are almost identical to the typing rules for PCF. However, all types are now randomized, and there is an extra rule for nodes.

[PredZero]	$\vdash (H \triangleright \text{pred } (0), bs = 0 : \text{rc } nat)$
[PredSucc]	$\vdash (H \triangleright \text{pred } (\underline{n+1}), bs = \underline{n} : \text{rc } nat)$
[ZeroZero]	$\vdash (H \triangleright \text{zero? } (0), bs = \text{tt } : \text{rc } bool)$
[ZeroSucc]	$\vdash (H \triangleright \text{zero? } (\underline{n+1}), bs = \text{ff } : \text{rc } bool)$
[Succ]	$\vdash (H \triangleright \text{succ } (\underline{n}), bs = \underline{n+1} : \text{rc } nat)$
[Mu]	$\dfrac{H, x : t \vdash M : \text{rc } t}{\vdash (H \triangleright \mu x : t.M, bs = M[\mu x : t.M/x] : \text{rc } t, bs)}$
[Node]	$\dfrac{\vdash (H \triangleright L, bs = L' : \text{rc } t) \quad \vdash (H \triangleright R, bs = R' : \text{rc } t, bs)}{\vdash (H \triangleright \text{Node } (L, R), bs = \text{Node } (L', R') : \text{rc } t}$
[Lambda]	$\dfrac{\vdash (H, x : s \triangleright M = N : \text{rc } t}{\vdash (H \triangleright \lambda x : s.M, bs = \lambda x : s.N : \text{rc } (s \to \text{rc } t), bs)}$
[Beta]	$\dfrac{H, x : s \vdash M : \text{rc } t \quad H \vdash \text{Leaf } N : \text{rc } s}{\vdash (H \triangleright (\lambda x : s.M)(\text{Leaf } N), bs = \texttt{getTree}(M[N/x], bs) : \text{rc } t, bs)}$
[Eta]	$\dfrac{H \vdash M : \text{rc } (s \to \text{rc } t) \quad x \notin Fv(M)}{\vdash (H \triangleright \lambda x : s.M(x), bs = \texttt{getTree}(M, bs) : \text{rc } (s \to \text{rc } t), bs)}$

Table 4.7: Equational Rules For rPCF

4.2.2 Equational Rules

We define a function `getTree` that takes a value, T of type rc a' and a list of booleans and returns a subtree of T. A list is of the form $[\texttt{hd}, \texttt{tl}]$, where `hd` is the first element of the list, and `tl` is the rest of the list. The list of booleans serves as our random oracle, determining which direction to move up the tree.

$$\texttt{getTree}(T, bs) = \begin{cases} T & \text{if } T = \text{Leaf } x \\ T & \text{if } bs \text{ is empty} \\ \texttt{getTree}(L, \texttt{tl}) & \text{if } T = \text{Node } (L, R), bs = [\text{ff}, \texttt{tl}] \\ \texttt{getTree}(R, \texttt{tl}) & \text{if } T = \text{Node } (L, R), bs = [\text{tt}, \texttt{tl}] \end{cases}$$

Evaluating terms of rPCF will use a list of booleans, bs, that begins as the empty list. The equational rules of rPCF are listed in Table 4.7 and Table 4.8.

Notice that for terms without random choice (without nodes), the equational rules exactly match the rules for normal PCF. For [Beta], [IfTrue], and [IfFalse], the `getTree` function gets called, but if the list of booleans, bs, is empty, this has no effect. The list of booleans only gets populated when determining the equality of terms with nodes.

[IfTrue] $$\frac{H \vdash M : \text{rc } t \qquad H \vdash N : \text{rc } t}{\vdash (H \triangleright \text{if tt then } M \text{ else } N, bs = \texttt{getTree}(M, bs) : \text{rc } t, bs)}$$

[IfFalse] $$\frac{H \vdash M : \text{rc } t \qquad H \vdash N : \text{rc } t}{\vdash (H \triangleright \text{if ff then } M \text{ else } N, bs = \texttt{getTree}(N, bs) : \text{rc } t, bs)}$$

[IfNode]

$$\frac{\vdash (H \triangleright \text{if } L \text{ then } M \text{ else } N, bs + \text{ff} = X) \qquad \vdash (H \triangleright \text{if } R \text{ then } M \text{ else } N, bs + \text{tt} = Y)}{\vdash (H \triangleright \text{if Node } (L, R) \text{ then } M \text{ else } N, bs = \text{Node } (X, Y), bs)}$$

[App] $$\frac{\vdash (H \triangleright M, bs = M' : \text{rc } (s \to \text{rc } t), bs) \qquad \vdash (H \triangleright N, bs = N' : \text{rc } s, bs)}{\vdash (H \triangleright M(N), bs = M'(N') : \text{rc } t, bs)}$$

[LeafNode]

$$\frac{\vdash (H \triangleright (\text{Leaf } F)(L), bs + \text{ff} = X) \qquad \vdash (H \triangleright (\text{Leaf } F)(R), bs + \text{tt} = Y)}{\vdash (H \triangleright (\text{Leaf } F)(\text{Node } (L, R)), bs = \text{Node } (X, Y), bs)}$$

[NodeLeaf]

$$\frac{\vdash (H \triangleright F(\text{Leaf } M), bs + \text{ff} = X) \qquad \vdash (H \triangleright G(\text{Leaf } M), bs + \text{tt} = Y)}{\vdash (H \triangleright (\text{Node } (F, G))(\text{Leaf } M), bs = \text{Node } (X, Y), bs)}$$

[NodeNode]

$$\frac{\vdash (H \triangleright F(L), bs + \text{ff} = X) \qquad \vdash (H \triangleright G(R), bs + \text{tt} = Y)}{\vdash (H \triangleright (\text{Node } (F, G))(\text{Node } (L, R)), bs = \text{Node } (X, Y), bs)}$$

[Cong]

$$\frac{\vdash (H \triangleright M, bs = N : \text{rc } nat, bs)}{\vdash (H \triangleright \text{pred } (M), bs = \text{pred } (N) : \text{rc } nat, bs)}$$

$$\frac{\vdash (H \triangleright M, bs = N : \text{rc } nat, bs)}{\vdash (H \triangleright \text{succ } (M), bs = \text{succ } (N) : \text{rc } nat, bs)}$$

$$\frac{\vdash (H \triangleright M, bs = N : \text{rc } nat, bs)}{\vdash (H \triangleright \text{zero? } (M), bs = \text{zero? } (N) : \text{rc } bool, bs)}$$

$$\frac{\vdash (H \triangleright M, bs = N : \text{rc } t, bs)}{\vdash (H \triangleright \mu x : t.M, bs = \mu x : t.N : \text{rc } t, bs)}$$

$$\frac{\vdash (H \triangleright L, bs = L' : \text{rc } bool, bs) \qquad \vdash (H \triangleright M, bs = M' : \text{rc } t, bs) \qquad \vdash (H \triangleright N, bs = N' : \text{rc } t, bs)}{\vdash (H \triangleright \text{if } L \text{ then } M \text{ else } N, bs = \text{if } L' \text{ then } M' \text{ else } N' : \text{rc } t, bs)}$$

Table 4.8: Equational Rules For rPCF (Continued)

4.2.3 Small Step Semantics

The transition rules for the small step semantics of rPCF are displayed in Table 4.9. Again, a list of booleans is used when evaluating a term. This list is never changed by a transition rule. However, when evaluating conditional statements or applications with a node, a false value, ff, is added to the list (denoted $bs + \text{ff}$) when evaluating the left subtree, and a true value, tt, is added (denoted $bs + \text{tt}$) when evaluating the right subtree. This is lifting the conditional and application operations to the randomized types, so the Kleisli extension of the monad is used. Note that when evaluating a single node, the Kleisli extension is not used, so nothing is added to the list.

Let $M \to^* N$ denote that N can be reached from M in zero or more steps.

Lemma 4.5. *If $M, bs \to^* N, bs$, then*

1. *$(Leaf\ F)(M), bs \to^* (Leaf\ F)(N), bs$, where $Leaf\ F$ can be pred , succ , zero? , or $\lambda x.N'$.*
2. *$M(L), bs \to^* N(L), bs$*
3. *if M then L_1 else $L_2, bs \to^*$ if N then L_1 else L_2, bs*
4. *$getTree(M, bs) \to^* getTree(N, bs)$*
5. *$(Node\ (F, G))(M), bs \to^* (Node\ (F, G))(N), bs$*

Proof. We outline the proof of 1. The proof is by induction on the length of the evaluation of $M, bs \to^* N, bs$. If the length is zero, $M \equiv N$, so the proof is trivial. For an evaluation of length n, we can write the evaluation as $M, bs \to M', bs \to^* N, bs$, where the second evaluation is of length $n - 1$. Then $(\text{Leaf } F)(M'), bs \to^* (\text{Leaf } F)(N), bs$ by the induction hypothesis, and $(\text{Leaf } F)(M), bs \to (\text{Leaf } F)(M'), bs$ by the small step semantics transition rules. Thus, $(\text{Leaf } F)(M), bs \to^* (\text{Leaf } F)(N), bs$. The proofs of $2 - 6$ are similar. ■

The following lemma can be proven similarly.

Lemma 4.6. *The following statements about the $\to^*$ relation are true:*

1. *If $F(Leaf\ N), bs + ff \to^* L$ and $G(Leaf\ N), bs + tt \to^* R$,*
   then $(Node\ (F, G))(Leaf\ N), bs \to^* Node\ (L, R), bs$
2. *If $(Leaf\ F)(L), bs + ff \to^* L'$ and $(Leaf\ F)(R), bs + tt \to^* R'$,*
   then $(Leaf\ F)(Node\ (L, R)), bs \to^* Node\ (L', R'), bs$

$$\frac{}{\text{pred } (0), bs \to 0} \qquad \frac{}{\text{pred } (\underline{n+1}), bs \to \underline{n}}$$

$$\frac{}{\text{zero? } (0), bs \to \text{tt}} \qquad \frac{}{\text{zero? } (\underline{n+1}), bs \to \text{ff}}$$

$$\frac{}{\text{if tt then } M \text{ else } N, bs \to \texttt{getTree}(M, bs), bs}$$

$$\frac{}{\text{if ff then } M \text{ else } N, bs \to \texttt{getTree}(N, bs), bs}$$

$$\frac{}{(\lambda x.M)(\text{Leaf } N), bs \to \texttt{getTree}(M[N/x], bs), bs}$$

$$\frac{M, bs \to M', bs}{\texttt{getTree}(M, bs), bs \to \texttt{getTree}(M', bs), bs}$$

$$\frac{}{\mu x.M, bs \to M[\mu x.M/x], bs}$$

$$\frac{M, bs \to M', bs}{\text{pred } (M), bs \to \text{pred } (M'), bs} \qquad \frac{M, bs \to M', bs}{\text{zero? } (M), bs \to \text{zero? } (M'), bs}$$

$$\frac{M, bs \to M', bs}{\text{succ } (M), bs \to \text{succ } (M'), bs} \qquad \frac{M, bs \to M', bs}{M(N), bs \to M'(N), bs}$$

$$\frac{N, bs \to N', bs}{(\text{Leaf } F)(N), bs \to (\text{Leaf } F)(N'), bs}$$

$$\frac{N, bs \to N', bs}{(\text{Node } (F, G))(N), bs \to (\text{Node } (F, G))(N'), bs}$$

$$\frac{F(\text{Leaf } N), bs + \text{ff} \to L \qquad G(\text{Leaf } N), bs + \text{tt} \to R}{(\text{Node } (F, G))(\text{Leaf } (N), bs \to \text{Node } (L, R), bs}$$

$$\frac{(\text{Leaf } F)(L), bs + \text{ff} \to L' \qquad (\text{Leaf } F)(R), bs + \text{tt} \to R'}{(\text{Leaf } F)(\text{Node } (L, R)), bs \to \text{Node } (L', R'), bs}$$

$$\frac{F(L), bs + \text{ff} \to L' \qquad G(R), bs + \text{tt} \to R'}{(\text{Node } (F, G))(\text{Node } (L, R)), bs \to \text{Node } (L', R'), bs}$$

$$\frac{L, bs \to L', bs}{\text{if } L \text{ then } M \text{ else } N, bs \to \text{if } L' \text{ then } M \text{ else } N, bs}$$

$$\frac{\text{if } L \text{ then } M \text{ else } N, bs + \text{ff} \to L' \qquad \text{if } R \text{ then } M \text{ else } N, bs + \text{tt} \to R'}{\text{if Node } (L, R) \text{ then } M \text{ else } N, bs \to \text{Node } (L', R'), bs}$$

$$\frac{L, bs \to L' \qquad R, bs \to R'}{\text{Node } (L, R), bs \to \text{Node } (L', R'), bs}$$

Table 4.9: Small Step Semantics For PCF

3. *If* $F(L), bs + ff \to^* L'$ *and* $G(R), bs + tt \to^* R'$,
   then $(Node\ (F, G))(Node\ (L, R)), bs \to^* Node\ (L', R'), bs$

4. *If if* L *then* M *else* $N, bs + ff^* L'$ *and if* R *then* M *else* $N, bs + tt \to^* R'$,
   then if $Node\ (L, R)$ *then* M *else* $N, bs \to^* Node\ (L', R'), bs$

5. *If* $L, bs \to^* L'$ *and* $R, bs \to^* R'$,
   then $Node\ (L, R), bs \to^* Node\ (L', R')$

As was stated for PCF, the small step semantics should match the typing and equational rules.

Theorem 4.7. *If* $H \vdash M : t$ *and* $M, bs \to N, bs$, *then* $H \vdash N : t$ *and* $\vdash (H \triangleright M, bs = N : t, bs)$.

Proof. The proof is by structural induction on M. If $M \equiv \text{pred}\ (0)$, then $N \equiv 0$. M and N are both of type rc *nat*, and by [PredZero], $\text{pred}\ (0), bs = 0$. If $M \equiv \text{pred}\ (\underline{n+1})$, then $N \equiv \underline{n}$. They are both of type rc *nat*, and by [PredSucc], $\text{pred}\ (\underline{n+1}), bs = \underline{n}$. If $M \equiv \text{zero?}\ (0)$, then $N \equiv \text{tt}$, they are both of type rc *bool*, and are equal by [ZeroZero]. If $M \equiv \text{zero?}\ (\underline{n+1})$, then $N \equiv \text{ff}$, they are both of type rc *bool*, and are equal by [ZeroSucc]. If $M \equiv \text{if tt then } N_1 \text{ else } N_2$, then $N \equiv \texttt{getTree}(N_1, bs)$, they have the same type, and by [IfTrue] are equal. The case where $M \equiv \text{if ff then } N_1 \text{ else } N_2$ is similar. If $M \equiv (\lambda x.V)(\text{Leaf } W)$, then $N \equiv getTree(V[W/x], bs)$. They both have the type of V and by [Beta] are equal. If $M \equiv \mu x.V$, then $N \equiv V[\mu x.V/x]$. They both have the type of V and by [Mu] are equal.

If $M \equiv \text{pred}\ (M')$, then $N \equiv \text{pred}\ (N')$, where $M', bs \to N', bs$. By the induction hypothesis, $M', bs = N', bs$, and by [Cong], $\text{pred}\ (M'), bs = \text{pred}\ (N'), bs$. The cases for $M \equiv \text{zero?}\ (M')$ and $M \equiv \text{succ}\ (M)$ are similar. If $M \equiv F(V)$ and $N \equiv F'(V)$, where $F, bs \to F', bs$, then by the induction hypothesis, $F = F'$, and by [App], $F(V) = F'(V)$. The remaining cases are all similar. ■

4.2.4 Big Step Semantics

The rules for the big step semantics of rPCF are displayed in Table 4.10. Again, only values appear on the right side of the ($\Downarrow$) operator. In PCF, values are natural numbers, booleans, or λ functions. In rPCF, values are nodes and leaves only containing values of PCF.

Now we show that the big and small step semantics coincide.

Theorem 4.8. $M, bs \Downarrow V \iff M, bs \to^* V$

$$\frac{}{0 \Downarrow 0} \qquad \frac{}{\lambda x.M \Downarrow \lambda x.M}$$

$$\frac{0 \Downarrow 0}{\mathrm{tt} \Downarrow \mathrm{tt}} \qquad \frac{\lambda x.M \Downarrow \lambda x.M}{\mathrm{ff} \Downarrow \mathrm{ff}}$$

$$\frac{M, bs \Downarrow 0}{\mathrm{pred}\ (M), bs \Downarrow 0} \qquad \frac{M, bs \Downarrow \underline{n+1}}{\mathrm{pred}\ (M), bs \Downarrow \underline{n}}$$

$$\frac{M, bs \Downarrow \underline{n}}{\mathrm{succ}\ (M), bs \Downarrow \underline{n+1}} \qquad \frac{M[\mu x.M/x], bs \Downarrow V}{\mu x.M, bs \Downarrow V}$$

$$\frac{M, bs \Downarrow 0}{\mathrm{zero?}\ (M), bs \Downarrow \mathrm{tt}} \qquad \frac{M, bs \Downarrow \underline{n+1}}{\mathrm{zero?}\ (M), bs \Downarrow \mathrm{ff}}$$

$$\frac{M, bs \Downarrow \mathrm{Node}\ (L, R) \quad \mathrm{zero?}\ (L), bs \Downarrow X \quad \mathrm{zero?}\ (R), bs \Downarrow Y}{\mathrm{zero?}\ (M), bs \Downarrow \mathrm{Node}\ (X, Y)}$$

$$\frac{M, bs \Downarrow \mathrm{Node}\ (L, R) \quad \mathrm{pred}\ (L), bs \Downarrow X \quad \mathrm{pred}\ (R), bs \Downarrow Y}{\mathrm{pred}\ (M), bs \Downarrow \mathrm{Node}\ (X, Y)}$$

$$\frac{M, bs \Downarrow \mathrm{Node}\ (L, R) \quad \mathrm{succ}\ (L), bs \Downarrow X \quad \mathrm{succ}\ (R), bs \Downarrow Y}{\mathrm{succ}\ (M), bs \Downarrow \mathrm{Node}\ (X, Y)}$$

$$\frac{M, bs \Downarrow \lambda x.M' \quad N \Downarrow \mathrm{Leaf}\ N' \quad M'[N'/x], bs \Downarrow V}{M(N), bs \Downarrow \mathtt{getTree}(V, bs)}$$

$$\frac{M, bs \Downarrow \mathrm{Node}\ (F, G) \quad N, bs \Downarrow \mathrm{Leaf}\ V \quad F(\mathrm{Leaf}\ V), bs + \mathrm{ff} \Downarrow X \quad G(\mathrm{Leaf}\ V), bs + \mathrm{tt} \Downarrow Y}{M(N), bs \Downarrow \mathrm{Node}\ (X, Y)}$$

$$\frac{M, bs \Downarrow \mathrm{Leaf}\ F \quad N \Downarrow \mathrm{Node}\ (L, R) \quad (\mathrm{Leaf}\ F)L, bs + \mathrm{ff} \Downarrow X \quad (\mathrm{Leaf}\ F)R, bs + \mathrm{tt} \Downarrow Y}{M(N), bs \Downarrow \mathrm{Node}\ (X, Y)}$$

$$\frac{M, bs \Downarrow \mathrm{Node}\ (F, G) \quad N \Downarrow \mathrm{Node}\ (L, R) \quad FL, bs + \mathrm{ff} \Downarrow X \quad FR, bs + \mathrm{tt} \Downarrow Y}{M(N), bs \Downarrow \mathrm{Node}\ (X, Y)}$$

$$\frac{B \Downarrow \mathrm{Leaf}\ \mathrm{tt} \qquad M, bs \Downarrow V}{\mathrm{if}\ B\ \mathrm{then}\ M\ \mathrm{else}\ N, bs \Downarrow \mathtt{getTree}(V, bs)}$$

$$\frac{B \Downarrow \mathrm{Leaf}\ \mathrm{ff} \qquad N, bs \Downarrow V}{\mathrm{if}\ B\ \mathrm{then}\ M\ \mathrm{else}\ N, bs \Downarrow \mathtt{getTree}(V, bs)}$$

$$\frac{B \Downarrow \mathrm{Node}\ (L, R) \quad \mathrm{if}\ L\ \mathrm{then}\ M\ \mathrm{else}\ N, bs + \mathrm{ff} \Downarrow X \quad \mathrm{if}\ R\ \mathrm{then}\ M\ \mathrm{else}\ N, bs + \mathrm{tt} \Downarrow Y}{\mathrm{if}\ B\ \mathrm{then}\ M\ \mathrm{else}\ N, bs \Downarrow \mathrm{Node}\ (X, Y)}$$

$$\frac{L, bs \Downarrow X \qquad R, bs \Downarrow Y}{\mathrm{Node}\ (L, R), bs \Downarrow \mathrm{Node}\ (X, Y)}$$

$$\frac{M, bs \Downarrow V}{\mathtt{getTree}(M, bs) \Downarrow \mathtt{getTree}(V, bs)}$$

Table 4.10: Big Step Semantics For rPCF

Proof. First, the proof of ($\Rightarrow$) is done by induction on the height of the derivation of $M \Downarrow V$. If the height is one, then M is a value, so $M \equiv V$. Now for derivations of larger height, the last step is considered. If $M \equiv \text{pred } (M')$ and $M, bs \Downarrow 0$ since $M', bs \Downarrow 0$, then $M', bs \to^* 0$ by the induction hypothesis, and by Lemma 4.5, $M, bs \equiv \text{pred } (M'), bs \to^* \text{pred } (0), bs \to 0$. The other cases for pred, succ, and zero? are similar for cases not involving nodes. For recursion, if $M \equiv \mu x.M'$, and $M, bs \Downarrow V$ since $M'[\mu x.M'/x], bs \Downarrow V$, then $M'[\mu x.M'/x], bs \to^* V$ by the induction hypothesis. $\mu x.M', bs \to M'[\mu x.M'/x], bs$ is one of the transition rules, so $M, bs \to^* V$.

Suppose $M \equiv F(N)$ and $M, bs \Downarrow \texttt{getTree}(V, bs) \equiv V'$ is derived using $(F, bs \Downarrow \lambda x.M')$, $(N, bs \Downarrow \text{Leaf } N')$, and $(M'[N'/x], bs \Downarrow V)$. Then by the induction hypothesis, $(F, bs \to^* \lambda x.M')$, $(N, bs \to^* \text{Leaf } N')$, and $(M'[N'/x], bs \to^* V)$. $F \equiv \lambda x.M'$ since there are no derivations of $\lambda x.M'$. Because $N, bs \to^* \text{Leaf } N'$, by Lemma 4.5, $F(N), bs \to^* F(\text{Leaf } N'), bs$, and by the small step transition rules, $F(\text{Leaf } N'), bs \to \texttt{getTree}(M'[N'/x], bs)$. Finally by Lemma 4.5, since $(M'[N'/x], bs \to^* V)$, $\texttt{getTree}(M'[N'/x], bs) \to^* \texttt{getTree}(V, bs)$. Thus, $F(N) \to^* \texttt{getTree}(V, bs)$.

If $M \equiv M'(N)$ and $M, bs \Downarrow \text{Node } (X, Y)$ using $M', bs \Downarrow \text{Node } (F, G)$ and $N, bs \Downarrow \text{Leaf } V$, then by the induction hypothesis, $M', bs \to^* \text{Node } (F, G)$ and $N, bs \to^* \text{Leaf } V$. By Lemma 4.5, $M'(N), bs \to^* (\text{Node } (F, G))(N), bs$, and $(\text{Node } (F, G))(N), bs \to^* (\text{Node } (F, G))(\text{Leaf } V), bs$. Also by the induction hypothesis, $F(\text{Leaf } V), bs + \text{ff} \to^* X$ and $G(\text{Leaf } V), bs + \text{tt} \to^* Y$. Thus by Lemma 4.6, $(\text{Node } (F, G))(\text{Leaf } V) \to^* \text{Node } (X, Y), bs$. Therefore, $M, bs \to^* \text{Node } (X, Y)$. The rest of the cases are similar.

For ($\Leftarrow$), first it is shown that if $M, bs \to N, bs \Downarrow V$, then $M, bs \Downarrow V$. This is done by induction on the structure of M. For constant M, there is nothing to prove. If $M \equiv \text{pred } (M')$, then there are few possibilities for N. If $N \equiv \text{pred } (N')$, with $M', bs \to N', bs$, then there are few cases for N'. If $N', bs \Downarrow 0$, then $M', bs \Downarrow 0$ and $\text{pred } (M'), bs \Downarrow 0 \equiv V$. If $N', bs \Downarrow \underline{n+1}$, then $M', bs \Downarrow \underline{n+1}$ and $\text{pred } (M'), bs \Downarrow \underline{n} \equiv V$. If $N', bs \Downarrow \text{Node } (L, R)$, where $\text{pred } (L), bs \Downarrow X$ and $\text{pred } (R), bs \Downarrow Y$, then $M', bs \Downarrow \text{Node } (L, R)$ and $V \equiv \text{Node } (X, Y)$. Thus, $M, bs \equiv \text{pred } (M'), bs \Downarrow \text{Node } (X, Y)$.

If $M \equiv \text{pred } (0) \to 0 \equiv N$, then $V \equiv 0$ and $\text{pred } (0), bs \Downarrow 0$. If $M \equiv \text{pred } (\underline{n+1}) \to \underline{n} \equiv N$, then $V \equiv \underline{n}$ and $\text{pred } (\underline{n+1}), bs \Downarrow \underline{n}$. The cases for succ and zero? are similar.

If $M \equiv (\text{if tt then } M' \text{ else } N', bs) \to \texttt{getTree}(M', bs) \Downarrow V$, then $V \equiv \mathit{getTree}(V', bs)$, where $M', bs \Downarrow V'$. Thus, $(\text{if tt then } M' \text{ else } N', bs) \Downarrow \texttt{getTree}(V', bs) \equiv V$. The case for if ff is

similar. If $M \equiv (\text{if Node } (L,R) \text{ then } M' \text{ else } N', bs) \to \text{Node } (L',R'), bs \Downarrow V$, then by induction:

$$(L', bs + \text{ff } \Downarrow V_L) \Rightarrow (\text{if } L \text{ then } M' \text{ else } N', bs + \text{ff } \Downarrow V_L)$$
$$(R', bs + \text{tt } \Downarrow V_R) \Rightarrow (\text{if } R \text{ then } M' \text{ else } N', bs + \text{tt } \Downarrow V_R)$$

Then (if tt then M' else $N', bs) \Downarrow \text{Node } (V_L, V_R) \equiv V$.

For application, suppose $M \equiv (\lambda x.M')(\text{Leaf } N'), bs \to \texttt{getTree}(M'[N/x], bs) \Downarrow V$. Then $V \equiv \texttt{getTree}(V', bs)$, where $M'[N/x], bs \Downarrow V'$. Thus $(\lambda x.M')(\text{Leaf } N'), bs \Downarrow getTree(V', bs) \equiv V$. If $M \equiv (\text{Node } (F,G))(\text{Node } (L,R)), bs \to \text{Node } (L',R'), bs \Downarrow V$, then by induction,

$$(L', bs + \text{ff } \Downarrow V_L) \Rightarrow (F(L), bs + \text{ff } \Downarrow V_L)$$
$$(R', bs + \text{tt } \Downarrow V_R) \Rightarrow (G(R), bs + \text{tt } \Downarrow V_R)$$

Then (Node $(F,G))$(Node $(L,R)), bs \Downarrow$ Node $(V_L, V_R) \equiv V$. The other cases for application are similar.

If $M \equiv \mu x.M', bs \to M'[\mu x.M'/x], bs \Downarrow V$, then $M \Downarrow V$ by the recursion rule for $(\Downarrow)$. The rest of the cases are all similar.

Now, the proof can be finished by induction on the length of the evaluation of $M \to^* V$. If the length is zero, then the result is obvious. If $M \to^*_{n+1} V$, then there is some N such that $M \to N \to^*_n V$. By induction, $N \Downarrow V$, and from the inductive proof above, $M \Downarrow V$. ■

4.2.5 Denotational Semantics

Now we develop a denotation semantics for rPCF using the random choice monad. For deterministic types, we use the same domains that were used in PCF: $[\![nat]\!] = \mathbb{N}_\perp$, $[\![bool]\!] = \mathbb{B}_\perp$ and $[\![s \to t]\!] = [[\![s]\!] \to [\![t]\!]]$.

Associate to the randomized type rc *nat* the domain $RC(\mathbb{N}_\perp)$ (so $[\![\text{rc } nat]\!] = RC(\mathbb{N}_\perp)$). For rc *bool*, we use the domain $RC(\mathbb{B}_\perp)$, and for rc $(s \to \text{rc } t)$, we use the domain $RC[[\![s]\!] \to [\![\text{rc } t]\!]]$.

Now we define some Scott continuous functions that are used in the denotational semantics. Define function $[\![\text{succ}]\!] : \mathbb{N}_\perp \to RC(\mathbb{N}_\perp)$ by:

$$[\![\text{succ}]\!](n) = \begin{cases} \perp & \text{if } n = \perp \\ (\epsilon, \chi_{n+1}) & \text{else} \end{cases}$$

$$[\![H \triangleright x : \text{rc } t, bs]\!]\rho = \rho(x)$$
$$[\![H \triangleright \lambda x : u.M : \text{rc } (u \to \text{rc } v), bs]\!]\rho = (\epsilon, (d \mapsto [\![H, x : u \triangleright \texttt{getTree}(M, bs) : \text{rc } v]\!](\rho[x \mapsto d])))$$
$$[\![H \triangleright M(N) : \text{rc } t, bs]\!]\rho = [\![\texttt{ev}^\dagger]\!]([\![H \triangleright M : \text{rc } (s \to \text{rc } t), bs]\!]\rho, [\![H \triangleright N : \text{rc } s, \texttt{nil}]\!]\rho)$$
$$[\![H \triangleright \mu x : t.M : \text{rc } t, bs]\!]\rho = \texttt{fix}(d \mapsto [\![H, x : t \triangleright M : \text{rc } t, bs]\!]\rho[x \mapsto d])$$
$$[\![H \triangleright \text{Leaf } \underline{n} : \text{rc } nat, bs]\!]\rho = (\epsilon, \chi_n)$$
$$[\![H \triangleright \text{Leaf tt} : \text{rc } bool, bs]\!]\rho = (\epsilon, \chi_{\text{tt}})$$
$$[\![H \triangleright \text{Leaf ff} : \text{rc } bool, bs]\!]\rho = (\epsilon, \chi_{\text{ff}})$$
$$[\![H \triangleright \text{succ } (M) : \text{rc } nat, bs]\!]\rho = [\![\text{succ}]\!]^\dagger([\![H \triangleright M, bs]\!]\rho)$$
$$[\![H \triangleright \text{pred } (M) : \text{rc } nat, bs]\!]\rho = [\![\text{pred}]\!]^\dagger([\![H \triangleright M, bs]\!]\rho)$$
$$[\![H \triangleright \text{zero? } (M) : \text{rc } bool, bs]\!]\rho = [\![\text{zero?}]\!]^\dagger([\![H \triangleright M, bs]\!]\rho)$$
$$[\![H \triangleright \text{if } L \text{ then } M \text{ else } N, bs]\!]\rho = [\![\text{if}^\dagger]\!]([\![H \triangleright L, \texttt{nil}]\!]\rho, [\![H \triangleright \texttt{getTree}(M, bs)]\!]\rho, [\![H \triangleright \texttt{getTree}(N, bs)]\!]\rho)$$
$$[\![H \triangleright \text{Node } (L, R), bs]\!]\rho = [\![\text{Node}]\!]([\![H \triangleright L, bs]\!]\rho, [\![H \triangleright R, bs]\!]\rho)$$
$$[\![H \triangleright \texttt{getTree}(M, bs)]\!]\rho = [\![\texttt{getTree}]\!]([\![H \triangleright M, bs]\!]\rho, bs)$$

Table 4.11: Denotational Semantics For rPCF

Define function $[\![\text{pred}]\!] : \mathbb{N}_\perp \to RC(\mathbb{N}_\perp)$ by:

$$[\![\text{pred}]\!](n) = \begin{cases} \perp & \text{if } n = \perp \\ (\epsilon, \chi_0) & \text{if } n = 0 \\ (\epsilon, \chi_{n-1}) & \text{else} \end{cases}$$

Define function $[\![\text{zero?}]\!] : \mathbb{N}_\perp \to RC(\mathbb{B}_\perp)$ by:

$$[\![\text{zero?}]\!](n) = \begin{cases} \perp & \text{if } n = \perp \\ (\epsilon, \chi_{\text{tt}}) & \text{if } n = 0 \\ (\epsilon, \chi_{\text{ff}}) & \text{else} \end{cases}$$

Define function $[\![\text{if}]\!] : RC(\mathbf{T})^2 \to \mathbb{B}_\perp \to RC(\mathbf{T})$ by:

$$[\![\text{if}]\!](x, y)(b) = \begin{cases} \perp & \text{if } b = \perp \\ x & \text{if } b = \text{tt} \\ y & \text{else} \end{cases}$$

Now define $[\![\text{if}^\dagger]\!] : RC(\mathbb{B}_\perp) \times RC(\mathbf{T})^2 \to RC(\mathbf{T})$ by

$$[\![\text{if}^\dagger]\!](b, x, y) = ([\![\text{if}]\!](x, y))^\dagger(b)$$

Define $[\![\texttt{ev}^\dagger]\!] : RC([\mathbf{S} \to RC(\mathbf{T})]) \times RC(\mathbf{S}) \to RC(\mathbf{T})$ by

$$[\![\texttt{ev}^\dagger]\!](f, x) = (\lambda g : [\mathbf{S} \to RC(\mathbf{T})].g^\dagger(x))^\dagger(f)$$

Define $[\![\text{Node}]\!] : RC(\mathbf{T})^2 \to RC(\mathbf{T})$ by

$$[\![\text{Node}]\!]((M, f), (N, g)) = (0 * M \cup 1 * N, F)$$

where $F(0 * w) = f(w)$ and $F(1 * w) = g(w)$.
Finally, define $[\![\texttt{getTree}]\!] : RC(\mathbf{T}) \times \{0, 1\}^\infty \to RC(\mathbf{T})$ by

$$[\![\texttt{getTree}]\!]((M, f), w) = (\{v \mid w * v \in M\}, g)$$

where $g(v) = f(w * v)$. If $\{v \mid w * v \in M\}$ is empty, then the result is $(\epsilon, \chi_{f \circ \pi_M(w)})$. Let $[\![\texttt{tails}]\!](r) \equiv [\![\texttt{getTree}]\!](r, 0)$ and $[\![\texttt{heads}]\!](r) \equiv [\![\texttt{getTree}]\!](r, 1)$.

Lemma 4.9. (Substitution) *If $H \vdash N : s$ and $H, x : s \vdash M : t$, then*

$$[\![H \triangleright M[N/x] : t]\!]\rho = [\![H, x : s \triangleright M : t]\!]\rho[x \mapsto ([\![H \triangleright N : s]\!]\rho)]$$

.

Proof. The proof is by induction on the structure of M. If M is a simply a natural number or boolean, the proof is immediate. Let $e = [\![H \triangleright N : s]\!]\rho$. Consider the case where $M \equiv x$. Then $s \equiv t$ and

$$\begin{aligned} [\![H \triangleright x[N/x] : t]\!]\rho &= [\![H \triangleright N : t]\!]\rho \\ &= \rho[x \mapsto e](x) \\ &= [\![H, x : s \triangleright M : t]\!]\rho[x \mapsto e] \end{aligned}$$

If $M \equiv y$, where y is distinct from x, then

$$\begin{aligned}[\![H \triangleright y[N/x] : t]\!]\rho &= [\![H \triangleright y : t]\!]\rho \\ &= \rho(y) \\ &= [\![H, x : s \triangleright y : t]\!]\rho[x \mapsto e]\end{aligned}$$

Let $M \equiv \text{pred } (M')$. Then

$$\begin{aligned}[\![H \triangleright \text{pred } (M')[N/x]]\!]\rho &= [\![\text{pred}]\!]^\dagger([\![H \triangleright M'[N/x]]\!]\rho) \\ &= [\![\text{pred}]\!]^\dagger([\![H, x \triangleright M']\!]\rho[x \mapsto e]) \\ &= [\![H, x \triangleright \text{pred } (M')]\!]\rho[x \mapsto e]\end{aligned}$$

The cases for succ and zero? are similar. If $M \equiv \lambda y.M'$, then

$$\begin{aligned}[\![H \triangleright (\lambda y.M')[N/x]]\!]\rho &= [\![H \triangleright \lambda y.M'[N/x]]\!]\rho \\ &= (d \mapsto [\![H, y \triangleright M'[N/x]]\!]\rho[y \mapsto d]) \\ &= (d \mapsto [\![H, x, y \triangleright M']\!](\rho[x \mapsto e][y \mapsto d])) \\ &= [\![H, x \triangleright \lambda y.M']\!](\rho[x \mapsto e])\end{aligned}$$

If $M \equiv M'(N')$ then

$$\begin{aligned}[\![H \triangleright (M'(N'))[N/x]]\!]\rho &= [\![\texttt{ev}^\dagger]\!]([\![H \triangleright M'[N/x]]\!]\rho, [\![H \triangleright N'[N/x]]\!]\rho) \\ &= [\![\texttt{ev}^\dagger]\!]([\![H, x \triangleright M']\!]\rho[x \mapsto e], [\![H, x \triangleright N']\!]\rho[x \mapsto e]) \\ &= [\![H, x \triangleright M'(N')]\!]\rho[x \mapsto e]\end{aligned}$$

If $M \equiv \mu y.M'$ then

$$\begin{aligned}[\![H \triangleright \mu y.M'[N/x]]\!]\rho &= \texttt{fix}(d \mapsto [\![H, y \triangleright M'[N/x]]\!](\rho[y \mapsto d]) \\ &= \texttt{fix}(d \mapsto [\![H, x, y \triangleright M']\!](\rho[x \mapsto e][y \mapsto d]) \\ &= [\![H, x \triangleright \mu y.M']\!](\rho[x \mapsto e])\end{aligned}$$

If $M \equiv \text{if } B \text{ then } M' \text{ else } N'$ then

$$\begin{aligned}
&[\![H \triangleright (\text{if } B \text{ then } M' \text{ else } N')[N/x]]\!]\rho \\
&= [\![\text{if}^\dagger]\!]([\![H \triangleright B[N/x]]\!]\rho, [\![H \triangleright M'[N/x]]\!]\rho, [\![H \triangleright N'[N/x]]\!]\rho) \\
&= [\![\text{if}^\dagger]\!]([\![H, x \triangleright B]\!]\rho[x \mapsto e], [\![H, x \triangleright M']\!]\rho[x \mapsto e], [\![H, x \triangleright N']\!]\rho[x \mapsto e]) \\
&= [\![H, x \triangleright (\text{if } B \text{ then } M' \text{ else } N')]\!]\rho[x \mapsto e]
\end{aligned}$$

Finally, if $M \equiv \text{Node } (L, R)$, then

$$\begin{aligned}
[\![H \triangleright \text{Node } (L, R)[N/x]]\!]\rho &= [\![\text{Node}]\!]([\![H \triangleright L[N/x]]\!]\rho, [\![H \triangleright R[N/x]]\!]\rho) \\
&= [\![\text{Node}]\!]([\![H, x \triangleright L]\!]\rho[x \mapsto e], [\![H, x \triangleright R]\!]\rho[x \mapsto e]) \\
&= [\![H, x \triangleright \text{Node } (L, R)]\!]\rho[x \mapsto e]
\end{aligned}$$

■

Theorem 4.10. (Soundness) *If* $\vdash (H \triangleright M, bs = N : t, bs)$*, then* $[\![H \triangleright M : t, bs]\!] = [\![H \triangleright N : t, bs]\!]$.

Proof. The proof is by induction on the height of the derivation of $\vdash (H \triangleright M = N : t)$. First the base cases have to be checked.

[PredZero]

If $M \equiv \text{pred } (0)$ and $N \equiv \text{Leaf } 0$, then

$$\begin{aligned}
[\![\text{pred } (0)]\!]\rho &= [\![\text{pred}]\!]^\dagger((\epsilon, \chi_0)) \\
&= [\![\text{pred}]\!]^\dagger(\eta(0)) \\
&= [\![\text{pred}]\!](0) \\
&= (\epsilon, \chi_0) \\
&= [\![\text{Leaf } 0]\!]\rho
\end{aligned}$$

The cases from [PredSucc], [ZeroZero], [ZeroSucc], and [Succ] are similar.

[Mu]

Suppose $M \equiv \mu x.M'$ and $N \equiv M'[\mu x.M'/x]$. Let $f(d) = [\![H, x \triangleright M']\!]\rho[x \mapsto d]$. Then

$$\begin{aligned}
[\![H \triangleright \mu x.M']\!]\rho &= \texttt{fix}(f) \\
&= f(\texttt{fix}(f)) \\
&= [\![H, x \triangleright M']\!]\rho[x \mapsto [\![H \triangleright \mu x.M']\!]\rho] \\
&= [\![H \triangleright M'[\mu x.M'/x]]\!]\rho
\end{aligned}$$

where the last equality follows from the Substitution lemma.

[Node]

Suppose $M \equiv$ Node (L, R) and $N \equiv$ Node (L', R'), and their equality is derived from the equality of L and L' and the equality of R and R'. By the induction hypothesis, $[\![L]\!]\rho = [\![L']\!]\rho$ and $[\![R]\!]\rho = [\![R']\!]$. Thus,

$$\begin{aligned}
[\![\text{Node } (L, R)]\!]\rho &= [\![\text{Node}]\!]([\![L]\!]\rho, [\![R]\!]\rho) \\
&= [\![\text{Node}]\!]([\![L']\!]\rho, [\![R']\!]\rho) \\
&= [\![\text{Node } (L', R')]\!]\rho
\end{aligned}$$

[Lambda]

Suppose $M \equiv \lambda x.M', bs$ and $N \equiv \lambda x.N', bs$. By induction, $[\![M']\!] = [\![N']\!]$ Thus

$$\begin{aligned}
[\![H \triangleright \lambda x.M', bs]\!]\rho &= (d \mapsto [\![\texttt{getTree}]\!]([\![H, x \triangleright M']\!], bs)(\rho[x \mapsto d])) \\
&= (d \mapsto [\![\texttt{getTree}]\!]([\![H, x \triangleright N']\!], bs)(\rho[x \mapsto d])) \\
&= [\![H \triangleright \lambda x.N', bs]\!]\rho
\end{aligned}$$

[Beta]

Suppose that $M \equiv (\lambda x.M')(\text{Leaf } N'), bs$ and that $N \equiv getTree(M'[N'/x], bs)$. Now let

$f_{bs} \equiv (d \mapsto [\![\texttt{getTree}]\!]([\![H, x \triangleright M']\!](\rho[x \mapsto d]), bs))$. Then

$$\begin{aligned}
[\![H \triangleright (\lambda x.M')(\text{Leaf } N'), bs]\!]\rho &= [\![\texttt{ev}^\dagger]\!]([\![H \triangleright \lambda x.M', bs]\!]\rho, [\![H \triangleright \text{Leaf } N']\!]\rho) \\
&= [\![\texttt{ev}^\dagger]\!]((\epsilon, f_{bs}), [\![H \triangleright \text{Leaf } N']\!]\rho) \\
&= (\lambda g.g^\dagger([\![H \triangleright \text{Leaf } N']\!]\rho))^\dagger((\epsilon, f_{bs})) \\
&= (\lambda g.g^\dagger([\![H \triangleright \text{Leaf } N']\!]\rho))(f_{bs}) \\
&= f_{bs}^\dagger([\![H \triangleright \text{Leaf } N']\!]\rho) \\
&= [\![\texttt{getTree}]\!]([\![H \triangleright M'[N'/x]]\!], bs)
\end{aligned}$$

[Eta]

Suppose $M \equiv \lambda x.M'(x), bs$ and $N \equiv getTree(M, bs)$

$$\begin{aligned}
[\![\lambda x.M'(x), bs]\!]\rho &= [\![\texttt{ev}^\dagger]\!]([\![\lambda x.M', bs]\!]\rho, [\![x]\!]\rho) \\
&= [\![\texttt{ev}^\dagger]\!]((\epsilon, (d \mapsto [\![\texttt{getTree}]\!]([\![H, x \triangleright M]\!], bs)(\rho[x \mapsto d]))), \rho(x)) \\
&= (d \mapsto [\![\texttt{getTree}]\!]([\![H, x \triangleright M]\!], bs)(\rho[x \mapsto d]))(\rho(x)) \\
&= [\![\texttt{getTree}]\!]([\![H, x \triangleright M]\!], bs)(\rho[x \mapsto \rho(x)]) \\
&= [\![\texttt{getTree}]\!]([\![H \triangleright M]\!], bs)\rho
\end{aligned}$$

[IfTrue]

Suppose $M \equiv \text{if tt then } M' \text{ else } N'$ and $N \equiv M'$. Then

$$\begin{aligned}
[\![\text{if tt then } M' \text{ else } N']\!]\rho &= [\![\text{if}^\dagger]\!]((\epsilon, \chi_{\text{tt}}), [\![M']\!]\rho, [\![N']\!]\rho) \\
&= [\![\text{if}]\!]^\dagger((\epsilon, \chi_{\text{tt}}))([\![M']\!]\rho, [\![N']\!]\rho) \\
&= [\![\text{if}]\!]^\dagger(\eta(\text{tt}))([\![M']\!]\rho, [\![N']\!]\rho) \\
&= [\![\text{if}]\!](\text{tt})([\![M']\!]\rho, [\![N']\!]\rho) \\
&= [\![M']\!]\rho
\end{aligned}$$

The [IfFalse] is case is identical.

[IfNode]

Suppose that $M \equiv \text{if Node } (L, R) \text{ then } M' \text{ else } N', bs$ and that $N \equiv \text{Node } (X, Y)$. Now let

$m_{bs} = [\![\texttt{getTree}]\!]([\![M']\!], bs)\rho$ and $n_{bs} = [\![\texttt{getTree}]\!]([\![N']\!], bs)\rho$.

$$\begin{aligned}[\![\text{if Node } (L,R) \text{ then } M' \text{ else } N', bs]\!]\rho &= [\![\texttt{if}^\dagger]\!]([\![\text{Node } (L,R)]\!]\rho, m_{bs}, n_{bs}) \\ &= ([\![\texttt{if}]\!](m_{bs}, n_{bs}))^\dagger([\![\text{Node}]\!]([\![L]\!]\rho, [\![R]\!]\rho))\end{aligned}$$

Let $[\![L]\!]\rho \equiv (P, g)$. For the first component, the words that start with 0 are

$$\bigcup_{w \in \pi_1([\![L]\!]\rho)} \text{Min } {\uparrow}(0 * w) \cap {\uparrow}\pi_1 \circ ([\![\texttt{if}]\!](m_{bs}, n_{bs})) \circ g(w))$$

Bringing a zero to the front leaves

$$0 * \bigcup_{w \in \pi_1([\![L]\!]\rho)} \text{Min } {\uparrow}w \cap {\uparrow}\pi_1 \circ [\![\texttt{tails}]\!]([\![\texttt{if}]\!](m_{bs}, n_{bs})) \circ g(w))$$

For the second component:

$$\pi_2(([\![\texttt{if}]\!](m_{bs}, n_{bs}))^\dagger([\![\text{Node}]\!]([\![L]\!]\rho, [\![R]\!]\rho)))(0 * z) = (\pi_2 \circ ([\![\text{if}]\!](m_{bs}, n_{bs})) \circ g(z))(z)$$

By induction, $[\![\text{if } L \text{ then } M' \text{ else } N', bs + \text{ff}]\!] = [\![X]\!]$ and the first component of $[\![\text{Node } (X,Y)]\!]\rho$ is $0 * \pi_1([\![X]\!]\rho) \cup 1 * \pi_1([\![Y]\!]\rho)$.

$$\begin{aligned}0 * \pi_1([\![X]\!]\rho) &= 0 * \pi_1([\![\text{if } L \text{ then } M' \text{ else } N', bs + \text{ff}]\!]\rho) \\ &= 0 * \pi_1(([\![\texttt{if}]\!](m_{bs+\text{ff}}, n_{bs+\text{ff}}))^\dagger([\![L]\!]\rho)) \\ &= 0 * \bigcup_{w \in \pi_1([\![L]\!]\rho)} \text{Min } {\uparrow}w \cap {\uparrow}\pi_1 \circ ([\![\texttt{if}]\!](m_{bs+\text{ff}}, n_{bs+\text{ff}})) \circ g(w) \\ &= 0 * \bigcup_{w \in \pi_1([\![L]\!]\rho)} \text{Min } {\uparrow}w \cap {\uparrow}\pi_1 \circ [\![\texttt{tails}]\!]([\![\texttt{if}]\!](m_{bs}, n_{bs})) \circ g(w))\end{aligned}$$

For the second component,

$$\begin{aligned}\pi_2([\![\text{Node } (X,Y)]\!]\rho)(0*z) &= \pi_2([\![X]\!]\rho)(z)\\ &= \pi_2([\![\text{if } L \text{ then } M' \text{ else } N', bs + \text{ff}]\!]\rho)(z)\\ &= \pi_2(([\![\texttt{if}]\!](m_{bs}, n_{bs}))^\dagger(P,g)(z)\\ &= (\pi_2 \circ ([\![\texttt{if}]\!](m_{bs}, n_{bs})) \circ g(z))(z)\end{aligned}$$

[LeafNode]

Suppose that $M \equiv (\lambda x.M')(\text{Node } (L,R))$ and that $N \equiv \text{Node } (X,Y)$. Similar to above, let $f_{bs} \equiv (d \mapsto [\![\texttt{getTree}]\!]([\![H, x \triangleright M']\!](\rho[x \mapsto d]), bs))$. Now

$$\begin{aligned}[\![(\lambda x.M')(\text{Node } (L,R)), bs]\!]\rho &= [\![\texttt{ev}^\dagger]\!]([\![(\lambda x.M'), bs]\!]\rho, [\![\text{Node } (L,R)]\!]\rho)\\ &= [\![\texttt{ev}^\dagger]\!]((\epsilon, f_{bs}), [\![\text{Node}]\!]([\![L]\!]\rho, [\![R]\!]\rho))\\ &= (\lambda g.g^\dagger([\![\text{Node}]\!]([\![L]\!]\rho, [\![R]\!]\rho)))(f_{bs})\\ &= f_{bs}^\dagger([\![\text{Node}]\!]([\![L]\!]\rho, [\![R]\!]\rho))\end{aligned}$$

Now let's consider the first component, looking only at words beginning with 0 (which L but not R determines).

$$\bigcup_{w \in \pi_1([\![L]\!]\rho)} \text{Min} \uparrow(0*w) \cap \uparrow\pi_1 \circ f_{bs} \circ (\pi_2([\![L]\!]\rho))(w)$$

Bringing a zero out to the front leaves:

$$0 * \bigcup_{w \in \pi_1([\![L]\!]\rho)} \text{Min} \uparrow w \cap \uparrow\pi_1 \circ [\![\texttt{tails}]\!](f_{bs} \circ (\pi_2([\![L]\!]\rho))(w))$$

Let $[\![L]\!]\rho \equiv (P,g)$. For the second component:

$$\begin{aligned}\pi_2(f_{bs}^\dagger([\![\text{Node}]\!]([\![L]\!]\rho, [\![R]\!]\rho)))(0*z) &= (\pi_2 \circ f_{bs} \circ g(z))(0*z)\\ &= (\pi_2 \circ [\![\texttt{tails}]\!](f_{bs} \circ g(z)))(z)\end{aligned}$$

By induction, $[\![(\lambda x.M')(L), bs + \text{ff}]\!] = [\![X]\!]$ and the first component of $[\![\text{Node } (X,Y)]\!]\rho$ is $0 * \pi_1([\![X]\!]\rho) \cup 1 * \pi_1([\![Y]\!]\rho)$.

$$\begin{aligned}
0 * \pi_1([\![X]\!]\rho) &= 0 * \pi_1([\![(\lambda x.M')(L), bs + \mathrm{ff}]\!]\rho) \\
&= 0 * \pi_1(f^{\dagger}_{bs+\mathrm{ff}}([\![L]\!]\rho)) \\
&= 0 * \bigcup_{w \in \pi_1([\![L]\!]\rho)} \mathrm{Min}\ {\uparrow}w \cap {\uparrow}\pi_1 \circ f_{bs+\mathrm{ff}} \circ (\pi_2([\![L,]\!]\rho))(w) \\
&= 0 * \bigcup_{w \in \pi_1([\![L]\!]\rho)} \mathrm{Min}\ {\uparrow}w \cap {\uparrow}\pi_1 \circ [\![\texttt{tails}]\!](f_{bs} \circ (\pi_2([\![L,]\!]\rho))(w))
\end{aligned}$$

For the second component:

$$\begin{aligned}
\pi_2([\![\text{Node } (X, Y)]\!]\rho)(0 * z) &= \pi_2([\![X]\!]\rho)(z) \\
&= \pi_2(f^{\dagger}_{bs+\mathrm{ff}}([\![L]\!]\rho))(z) \\
&= (\pi_2 \circ f_{bs+\mathrm{ff}} \circ g(z))(z) \\
&= (\pi_2 \circ [\![\texttt{tails}]\!](f_{bs} \circ g(z)))(z)
\end{aligned}$$

The words beginning with 1 are similar, using R instead of L.

[NodeLeaf]

Suppose that $M \equiv \text{Node } (F, G)(\text{Leaf } N')$ and $N \equiv \text{Node } (X, Y)$. Then

$$\begin{aligned}
[\![\text{Node } (F, G)(\text{Leaf } N')]\!]\rho &= [\![\texttt{ev}^{\dagger}]\!]([\![(\text{Node } (F, G), bs]\!]\rho, [\![\text{Leaf } N']\!]\rho) \\
&= [\![\texttt{ev}^{\dagger}]\!]([\![\text{Node}]\!]([\![F, bs]\!]\rho, [\![G, bs]\!]\rho), (\epsilon, \chi_{\underline{n}})) \\
&= (\lambda g.g^{\dagger}(\epsilon, \chi_{\underline{n}}))^{\dagger}([\![\text{Node}]\!]([\![F, bs]\!]\rho, [\![G, bs]\!]\rho)) \\
&= (\lambda g.g(\underline{n}))^{\dagger}([\![\text{Node}]\!]([\![F, bs]\!]\rho, [\![G, bs]\!]\rho))
\end{aligned}$$

Now consider the words beginning with 0 of the first component.

$$\bigcup_{w \in \pi_1([\![F, bs]\!]\rho)} \mathrm{Min}\ {\uparrow}(0 * w) \cap {\uparrow}\pi_1 \circ (\pi_2([\![F, bs]\!]\rho)(w))(\underline{n})$$

Bringing a zero to the front leaves:

$$0 * \bigcup_{w \in \pi_1([\![F, bs]\!]\rho)} \mathrm{Min}\ {\uparrow}w \cap {\uparrow}\pi_1 \circ [\![\texttt{tails}]\!](\pi_2([\![F, bs]\!]\rho)(w))(\underline{n})$$

Let $[\![F, bs]\!]\rho \equiv (P, f_{bs})$. For the second component:

$$\pi_2([\![\text{Node } (F,G)(\text{Leaf } N')]\!]\rho)(0*z) = (\pi_2((f_{bs}(z))(\underline{n}))(0*z)$$
$$= (\pi_2([\![\texttt{tails}]\!]((f_{bs}(z))(\underline{n})))(z))$$

By induction, $[\![F(\text{Leaf } N'), bs + \text{ff}]\!] = [\![X]\!]$.

$$\begin{aligned}
0 * \pi_1([\![X]\!]\rho) &= 0 * \pi_1([\![F(\text{Leaf } N'), bs + \text{ff}]\!]\rho) \\
&= 0 * \pi_1((\lambda g.g(\underline{n}))^\dagger([\![F, bs + \text{ff}]\!]\rho) \\
&= 0 * \bigcup_{w \in \pi_1([\![F, bs+\text{ff}]\!]\rho)} \text{Min } {\uparrow}w \cap {\uparrow}\pi_1 \circ (\pi_2([\![F, bs + \text{ff}]\!]\rho)(w))(\underline{n}) \\
&= 0 * \bigcup_{w \in \pi_1([\![F, bs]\!]\rho)} \text{Min } {\uparrow}w \cap {\uparrow}\pi_1 \circ [\![\texttt{tails}]\!](\pi_2([\![F, bs]\!]\rho)(w))(\underline{n})
\end{aligned}$$

For the second component:

$$\begin{aligned}
\pi_2([\![\text{Node } (X,Y)]\!]\rho)(0*z) &= \pi_2([\![X]\!]\rho)(z) \\
&= (\pi_2((f_{bs+\text{ff}}(z))(\underline{n}))(0*z) \\
&= (\pi_2([\![\texttt{tails}]\!]((f_{bs}(z))(\underline{n})))(z))
\end{aligned}$$

[NodeNode]

Suppose $M \equiv (\text{Node } (F,G))(\text{Node } (L,R))$ and $N \equiv \text{Node } (X,Y)$.

$$\begin{aligned}
[\![\text{Node } (F,G)(\text{Node } (L,R))]\!]\rho &= [\![\texttt{ev}^\dagger]\!]([\![(\text{Node } (F,G), bs]\!]\rho, [\![\text{Node } (L,R)]\!]\rho) \\
&= (\lambda g.g^\dagger([\![\text{Node}]\!]([\![L]\!]\rho, [\![R]\!]\rho)))^\dagger([\![\text{Node}]\!]([\![F, bs]\!]\rho, [\![G, bs]\!]\rho))
\end{aligned}$$

Now consider the words beginning with 0 in the first component:

$$\bigcup_{w \in \pi_1([\![F, bs]\!]\rho)} \text{Min } {\uparrow}(0*w) \cap {\uparrow}\pi_1 \circ (\pi_2([\![F, bs]\!]\rho)(w))^\dagger([\![\text{Node}]\!]([\![L]\!]\rho, [\![R]\!]\rho))$$

Since we are only considering words beginning with 0, we can do the same for the second Kleisli extension to get

$$\bigcup_{w \in \pi_1([\![F, bs]\!]\rho)} \text{Min } {\uparrow}(0*w) \cap {\uparrow}(\bigcup_{z \in \pi_1([\![L]\!]\rho)} \text{Min } {\uparrow}(0*z) \cap {\uparrow}(\pi_1 \circ (\pi_2([\![F, bs]\!]\rho)(w))(z)))$$

Bringing a zero to the front gives:

$$0 * \bigcup_{w \in \pi_1([\![F,bs]\!]\rho)} \text{Min } \uparrow w \cap \uparrow(\bigcup_{z \in \pi_1([\![L]\!]\rho)} \text{Min } \uparrow z \cap \uparrow(\pi_1 \circ [\![\texttt{tails}]\!](\pi_2([\![F,bs]\!]\rho)(w))(z)))$$

By induction, $[\![F(L), bs + \text{ff}]\!] = [\![X]\!]$.

$$\begin{aligned}
0 * \pi_1([\![X]\!]\rho) &= 0 * \pi_1([\![F(L), bs + \text{ff}]\!]\rho) \\
&= 0 * \pi_1((\lambda g.g^\dagger([\![L]\!]\rho))^\dagger([\![F, bs + \text{ff}]\!]\rho)) \\
&= 0 * \bigcup_{w \in \pi_1([\![F,bs+\text{ff}]\!]\rho)} \text{Min } \uparrow w \cap \uparrow \pi_1 \circ (\pi_2([\![F, bs + \text{ff}]\!]\rho)(w))^\dagger([\![L]\!]\rho) \\
&= 0 * \bigcup_{w \in \pi_1([\![F,bs+\text{ff}]\!]\rho)} \text{Min } \uparrow w \cap \uparrow(\bigcup_{z \in \pi_1([\![L]\!]\rho)} \text{Min } \uparrow z \cap \uparrow(\pi_1 \circ \pi_2([\![F, bs + \text{ff}]\!]\rho)(w))(z))) \\
&= 0 * \bigcup_{w \in \pi_1([\![F,bs]\!]\rho)} \text{Min } \uparrow w \cap \uparrow(\bigcup_{z \in \pi_1([\![L]\!]\rho)} \text{Min } \uparrow z \cap \uparrow(\pi_1 \circ [\![\texttt{tails}]\!](\pi_2([\![F, bs]\!]\rho)(w))(z)))
\end{aligned}$$

The second component is similar to the previous two cases, but much messier since there is a nesting of the Kleisli extensions.

All of the [Cong] cases are very simple equations.

■

Corollary 4.11. *If $M \to N$, then $[\![M]\!] = [\![N]\!]$.*

Corollary 4.12. *If $M \Downarrow V$, then $[\![M]\!] = [\![V]\!]$.*

Theorem 4.13. (Adequacy) *If M is a closed term of ground type and $[\![M]\!] = [\![V]\!]$ for a value V, then $M \Downarrow V$.*

Proof. The proof is by induction on the structure of V. If V is a leaf (no nodes), then V is the embedding of a PCF term into rPCF. $[\![M]\!] = [\![V]\!] \equiv (\epsilon, \chi_v)$ for some v and there is a corresponding term M' in PCF such that $[\![M']\!] = v$. Since adequacy is proven for the semantics of PCF, $M' \Downarrow V'$ and thus, $M \Downarrow V$ in rPCF. Now if V is of the form Node (L, R), where $[\![M]\!] = [\![\text{Node } (L, R)]\!]$, then $[\![\texttt{getTree}]\!]([\![M]\!], \text{ff}) = L$ and $[\![\texttt{getTree}]\!]([\![M]\!], \text{tt}) = R$. By induction $getTree(M, \text{ff}) \Downarrow L$ and $getTree(M, \text{tt}) \Downarrow R$. Thus, $M \Downarrow \text{Node } (L, R)$. ■

Chapter 5

Implementation in Functional Programming

5.1 Functional Programming

The idea of a monad from category theory began to be applied to programming languages when Moggi developed a categorical semantics of computation using monads [42]. Wadler then used this idea to express features of functional programming languages using monads [26]. He generalized the functional programming structure of list comprehensions to a monad structure and showed that other programming features like exceptions and continuations could also be expressed using monads. Wadler was one of the main designers of the Haskell programming language, in which monads play a pivotal role.

Other functional programming languages may not explicitly use monads, but a programmer who thinks abstractly using monads can still make use of them. Different forms of monads have been implemented in languages such as Scala, Python, C#, OCaml, and many more. However, although monads can be defined in these languages, there is no way to verify that the defined construct obeys the monad laws. It can be checked that all of the relevant functions are defined and that everything is of the correct type, but there is no equational reasoning to prove that the required equations hold. To this end, an interactive theorem prover, like Isabelle, can be used.

5.1.1 Haskell

The random choice monads represent the possible results of coin flips by a binary tree. Developing an operational version of the monad should also contain some form of a binary tree. A monad for binary leafy trees (ones with data values only at leaves) is supported in Haskell. A possible implementation of this monad is shown in Figure 5.1. In Haskell, a new datatype can be declared with the keyword `data`. Here, given some type `a`, a new datatype called `Tree a` is created recursively. For example, `Tree Int` will be a binary tree of integers. The base case for this datatype is `Leaf a` which is a tree of height zero, only consisting of a root that contains some value in `a`. The recursive case says that given two binary trees, we can combine them into one bigger tree where each of the given trees is a subtree.

Now that the datatype is created, we can declare it to be a monad. Just as in category theory, to form a monad, we can define the unit and Kleisli extension. In Haskell, the unit is called `return` and is a function of type `a -> Tree a`. Here the unit takes an element of `a` and creates the tree of height zero, where the root contains that same element of `a`.

The Kleisli extension is denoted by `>>=` in Haskell and is a function with a type signature of `Tree a -> (a -> Tree b) -> Tree b`. It is an infix operation, so the element of `Tree a` goes

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)

instance Monad Tree where
   return = Leaf
   Leaf x >>= h = h x
   Branch l r >>= h = Branch (l >>= h) (r >>= h)
```

Figure 5.1: The monad of leafy trees

to the left and the function of type `a -> Tree b` goes to the right. It then outputs an element of `Tree b`. The Kleisli extension shown here moves up a tree until it reaches a leaf with value x. It then replaces x with the entire tree given by `h x`. This exactly mimics the behavior of the Kleisli extension used by Goubault-Larrecq and Varacca for their uniform continuous random variables.

Now to implement the RC monad in Haskell, the datatype will be defined as above.

```
data RC a = Leaf a | Node (RC a) (RC a)
```

If we wish to use the extended monad, RC',which uses Scott closed sets instead of the antichains, then we can supply values of `a` at every level of the binary tree instead of just the leaves. Here each node has a value of `a` along with its two subtrees.

```
data RC' a = Leaf a | Node a (RC' a) (RC' a)
```

A datatype can be declared as a functor by defining the functor map, called `fmap`. This has a type signature of `(a -> b) -> RC a -> RC b`. This simply leaves the structure of the binary tree alone and applies a function `f` to each value of `a`

```
instance Functor RC where
    fmap f (Leaf x) = Leaf (f x)
    fmap f (Node l r) = Node (fmap f l) (fmap f r)

instance Functor RC' where
    fmap f (Leaf x) = Leaf (f x)
    fmap f (Node x l r) = Node (f x) (fmap f l) (fmap f r)
```

Now we define some functions that will be useful in implementing the monad. First we define `getTree`, a function that uses a list of coin flips to move up a binary tree. If it reaches the top of the tree, a leaf, before using all the coin flips, then the remaining flips are disregarded and the leaf is returned. If it uses the coin flips without reaching the top, then it returns the remaining subtree. Here a coin flip is represented by an integer. A zero represents tails and any nonzero

integer represents heads.

```
getTree :: RC a -> [Int] -> RC a
getTree (Leaf x) _ = (Leaf x)
getTree t [] = t
getTree (Node l r) (0:xs) = getTree l xs
getTree (Node l r) (_:xs) = getTree r xs
```

The definition for `RC'` is almost identical. Also, for `RC'`, where every level of the trees has values, we can define a function `getValue` that uses a list of coin flips to pick out a value in the tree.

```
getValue :: RC' a -> [Int] -> a
getValue (Leaf x) _ = x
getValue (Node x _ _) [] = x
getValue (Node _ l r) (0:xs) = getValue l xs
getValue (Node _ l r) (_:xs) = getValue r xs
```

Now for the monad, the unit will be the same as above.

```
unit :: a -> RC a
unit x = Leaf x
```

For the Kleisli extension, we define a helper function `kleisli`. This is similar to the Kleisli extension defined above, except that as it moves up the tree, it keeps track of the coin flips needed to get to its location. Therefore, it needs another argument with a list of integers. Let `f` be function of type `a -> RC b` that is being extended. When `kleisli` gets to the top of the tree, which is of the form `Leaf x`, it applies `f` to `x` to get another tree, but it does not use this entire tree. Instead, it uses the tracked coin flips to traverse this new tree.

```
kleisli :: (a -> RC b) -> RC a -> [Int] -> RC b
kleisli f (Leaf x) xs = getTree (f x) xs
kleisli f (Node l r) xs = Node (kleisli f l (xs++[0]))
                               (kleisli f r (xs++[1]))
```

For `RC'`, a slight change is needed to handle the intermediate values.

```
kleisli :: (a -> RC' b) -> RC' a -> [Int] -> RC' b
kleisli f (Leaf x) xs = getTree (f x) xs
kleisli f (Node x l r) xs = Node (getValue (f x) xs)
                                 (kleisli f l (xs++[0]))
                                 (kleisli f r (xs++[1]))
```

Now we can declare the monad instances.

```
instance Monad RC where
        return = unit
        m >>= f = kleisli f m []

instance Monad RC' where
        return = unit
        m >>= f = kleisli f m []
```

From now on, we will only work with `RC'`, so that our binary trees have values at all levels. Here we create two random choices of integers.

```
tree1 = Node 1 (Leaf 2) (Leaf 3)
tree2 = Node 5 (Leaf 6) (Node 7 (Leaf 8) (Leaf 9))
```

We use a function `showRC` to display these random choices. Using `showRC tree1` results in:

```
    2
1
    3
```

and `showRC tree2` outputs:

```
    6
5
        8
    7
        9
```

Instead of using the monad operations like `>>=` directly, Haskell has a `do` notation that provides syntactic sugar when dealing with monadic elements. For example, to double every value of `tree1`, we can use the following code:

```
tree3 = do
        i <- tree1
        return (2*i)
```

Then `tree3` would be the random choice represented by:

```
    4
2
    6
```

To lift the addition operation on integers to an operation on random choices of integers, we can write:

```
tree4 = do
        i <- tree1
        j <- tree2
        return (i+j)
```

This results in the random choice:

```
    8

6

        11

    10

        12
```

In Section 2.4.2, it was shown how the Kleisli extension lifts a binary operation. The behavior shown here is same as was described in that section.

To make a random choice, we can use the standard Random library in Haskell. To simulate flipping a coin, we first make a coin datatype.

```
data Coin = Heads | Tails
```

We can declare `Coin` to be an instance of `Random` by defining a function that maps a random choice of a number to a random choice of `Heads` or `Tails`.

```
instance Random Coin where
  random g = case random g of
               (r,g') | r < 1/2   = (Tails, g')
                      | otherwise = (Heads, g')
```

Here we a given a random choice of a number between zero and one. By using `1/2`, we create a fair coin. This can be changed to produce a biased coin.

Now we use this to create a function `choose :: RC' a -> Int -> IO a` that when given a tree `t` and nonnegative integer `n`, it uses `n` coin flips to traverse `t` and outputs the value at its final location. Notice that the output is of type `IO a` instead of just `a`. The IO monad in Haskell contains actions that use input or output. In this case, the act of getting a random value is viewed as input to the rest of the function. Therefore, the output must be contained in the IO monad.

```
choose :: RC' a -> Int -> IO a
choose (Leaf _ x) _  = return x
choose (Node _ x _ _) 0 = return x
choose (Node _ x l r) n = do
                    c <- randomIO
                    (if (c==Heads) then (choose r (n-1))
                                   else (choose l (n-1)))
```

Using this function on our first tree, `choose tree1 0` will result in the IO action that always produces the value `1`. No coins are flipped, so the function just stays at the root of the tree. Using a positive number of coin flips, `choose tree1 1` will now result in the IO action that produces the values `2` and `3` each with equal probability.

As described above, using the Kleisli extension to add two random numbers will not choose the two numbers independently of one another. Each coin flip will be used in both random choices. However, if it is desired that two independent choices be made, this can still be done by making the random choices before performing the addition.

```
sequentialAdd = do
        i <- (choose tree1 1)
        j <- (choose tree2 2)
        return (i+j)
```

Here, `i` and `j` are in the IO monad, so the addition uses its Kleisli extension, which performs actions sequentially. Therefore, one coin flip is used to choose a number from `tree1`, then two more coin flips are used to choose a number from `tree2`. Finally, the two numbers get added together.

It is even possible to perform the independent random choices concurrently, using Haskell's `Async` library.

```
concurrentAdd = do
        (i,j) <- concurrently (choose tree1 1) (choose tree2 2)
        return (i+j)
```

Here, the random choices are made concurrently, but the coin flips are made independently.

5.1.2 Scala

The Scala programming language is an object-oriented and functional programming language designed to be compiled and executed on a Java virtual machine. Unlike Haskell, Scala does not use category theory terms like functor and monad by name, but its type system still allows for the creation of such objects. Similar to Haskell's `do` notation, Scala has `for` expressions that provide syntactic sugar for monad-like objects.

In Scala, the random choice functor can be implemented as a trait, with

```
trait RC[+A]
```

The trait can be thought of as a functor that acts on the category of types. `A` is a generic type, and `+A` means that the functor is covariant. This trait consists of a tree with values of type `A`, and the tree is defined recursively with nodes and leaves.

```
case class Node[A](value: A, left: RC[A], right: RC[A]) extends RC[A]
case class Leaf[A](value: A) extends RC[A]
```

To describe how the functor acts on functions, the `map` function is defined. Just as in the *RC* functor, the tree structure is not changed, so a leaf stays a leaf, and node stays a node.

```
def map[B](f: A => B): RC[B] = this match {
    case l: Leaf[A] => Leaf(f(l.value))
    case n: Node[A] => Node(f(n.value), n.left map f, n.right map f)
}
```

The unit of the monad is given as follows:

```
private def unit[B] (value: B) = Leaf(value)
```

This simply takes a value of a base type B and creates a leaf containing that value. This corresponds to $\eta(d) = (\epsilon, \chi_d)$.

To define the Kleisli extension of the monad, two helper functions are first defined. The function `getValue` takes a list of zeroes and ones (which represents a word in $\{0,1\}^\infty$) and gets the value stored at that word. The function `getTree` also takes a word of zeroes and ones and gets the section of the tree above that word.

```
def getValue(word: List[Int]): A = this match {
    case n: Node[A] => if (word == List()) n.value
            else if (word.head == 0) n.left.getValue(word.tail)
            else n.right.getValue(word.tail)
    case l: Leaf[A] => l.value
}

def getTree(word: List[Int]): RV[A] = this match {
    case n: Node[A] => if (word == List()) n
            else if (word.head == 0) n.left.getTree(word.tail)
            else n.right.getTree(word.tail)
    case l: Leaf[A] => l
}
```

Now given a function `f:A => RV[B]`, the Kleisli extension, called `flatMap` in Scala, must take an object in `RV[A]` and return an object in `RV[B]`.

```
def flatMap[B](f: A => RV[B]): RV[B] = {
  def kleisli(t: RV[A], word: List[Int]): RV[B] = {
    t match {
      case l: Leaf[A] => f(l.value).getTree(word)
      case n: Node[A] => Node(f(n.value).getValue(word),
                              kleisli(n.left, word ++ List(0)),
                              kleisli(n.right, word ++ List(1)))
      }
  }
  kleisli(this, List())
}
```

The important thing to notice here is that when moving up the tree, a zero or one is added to `word` depending on the direction taken. Then when a leaf is reached, the function `f` is applied to get an object of `RV[B]`. However, only the portion of that tree above `word` is used. This is what differentiates this Kleisli extension from the original attempt at one.

With the Kleisli extension defined, the multiplication of the monad, called `flatten` in Scala, can easily be defined.

```
def flatten[B] (tree: RV[RV[B]]): RV[B] = tree flatMap (x => x)
```

Now Scala's `for` expressions can be used with these objects. Let's create two trees containing integers:

```
val tree1:RV[Int] = Node(1, Leaf(2), Leaf(3));
```

This can be displayed as:

```
   2
1
   3
```

For the second tree:

```
val tree2:RV[Int] = Node(5, Leaf(6), Node(7, Leaf(8), Leaf(9)));
```

This can be displayed as:

```
    6
5
        8
    7
        9
```

The binary operation of addition on integers can be lifted to these trees of integers.

```
val tree3 = for (i <- tree1; j <- tree2) yield i+j
```

which results in:

```
    8
6
        11
    10
        12
```

For a randomized algorithm, only one branch of the tree should be traversed, in a random direction. Scala's built in `Random` library can be used as the oracle needed.

```
private val oracle = new Random()

def choose(num: Int): A = this match {
    case l: Leaf[A] => l.value
    case n: Node[A] => {
        if (num==0) n.value
        else if (oracle.nextBoolean) n.right.choose(num-1)
        else n.left.choose(num-1)
    }
}
```

Now calling `tree3.choose(2)` will return 8 with $\frac{1}{2}$ probability or 11 or 12 with $\frac{1}{4}$ probability.

5.1.3 Isabelle

As stated above, we can define monads in programming languages like Haskell and Scala; however, there is no way to ensure that our definition obeys the monad laws. To this end, we can use an interactive theorem prover like Isabelle.

Isabelle is a proof assistant based off of LCF, Scott's logic of computable functions (PCF is a programming language based on LCF). Isabelle provides a functional programming language along

with a logical system that can be used to prove properties of programs (Isabelle can actually be used to express and prove theorems about many mathematical concepts, but we are only concerned with the functional programming aspect).

Just like in Haskell and Scala, we can write programs to define new datatypes and functions. First, we define our datatype.

```
datatype 'a rc =
  Leaf 'a |
  Node 'a "'a rc" "'a rc"
```

We then create the functions `getValue` and `getTree`, whose (omitted) definitions are nearly identical to the analogous Haskell functions. Now, some simple lemmas can be proven in Isabelle.

```
lemma: "getTree t ys = Leaf a ⟹ getTree t (ys @ b) = Leaf a"
  apply (induct t ys rule: getTree.induct)
  apply simp_all
  done
```

This lemma states that if using the `getTree` function results in a leaf, then calling the same function but with extra bits will produce the same result. A leaf is returned if the tree traversal reaches the end of the tree. In this case, adding more bits will not change anything since there is nowhere else to go. In Isabelle, this can be proven by structural induction, using the definition of `getTree`.

When proving a theorem in Isabelle, there is always a current goal that needs to be proved. At the beginning, the desired theorem is the goal. In the previous lemma, the first step involves structural induction. This replaces the goal with subgoals containing the base and inductive cases. Then using `apply simp_all` attempts to simplify each goal by using any relevant definitions. For this lemma, this is all that is needed to finish the proof.

We can prove two more simple lemmas about the two functions.

```
lemma: "getTree t (xs @ ys) = getTree (getTree t xs) ys"
  apply (induction t xs rule:getTree.induct)
  apply auto
  done

lemma: "getValue t (xs @ ys) = getValue (getTree t xs) ys"
  apply (induct t xs rule: getTree.induct)
  apply auto
  done
```

Again, structural induction is used to prove these two lemmas. The proofs are finished by using `apply auto`. This uses built-in automated tools to look for a solution. Unfortunately, this only

works for very basic proofs.

To prove the monad laws, the unit and Kleisli extension must be defined.

```
fun unit :: "'a ⇒ 'a rc" where
  "unit x = Leaf x"

fun flatMap :: "('a ⇒ 'b rc) ⇒ 'a rc ⇒ 'b rc" where
  "flatMap f x = kleisli f x []"
```

The function `kleisli` is defined as it was in Haskell. Now, the monad laws can be proved.

```
lemma "flatMap f (unit x) = f x"
  by simp

lemma "(flatMap unit) x = x"
  by auto

lemma "(flatMap k) ((flatMap h) t) =
       (flatMap (λx. (flatMap k) (h x))) t"
```

The first two monad laws are trivially proven. The proof of the third monad law is omitted here and is much more involved. Its proof can be found in the included source code.

As stated above, the implementation of the monad in functional languages like Haskell and Scala does not verify that the monad laws are obeyed. Furthermore, the order structure on the objects on which the monad is defined is not explicitly present. However, we can define the order in Isabelle and use it to prove that the Kleisli extension is monotone.

We start by importing the `Porder` theory which has definitions and some simple lemmas proven about partial orders. A partial order must have a defined order relation, `below`, which can be represented with the infix operator $\sqsubseteq$. Here we define the order relation.

```
fun myBelow :: "'a::below rc ⇒ 'a rc ⇒ bool" where
  "myBelow (Leaf x) (Leaf y) = below x y" |
  "myBelow (Leaf x) (Node y l r) = below x y" |
  "myBelow (Node x l r) (Leaf y) = False" |
  "myBelow (Node x l r) (Node y l2 r2) =
    (below x y ∧ myBelow l l2 ∧ myBelow r r2)"
```

Now we use `myBelow` to instantiate the order relation.

```
instantiation rc :: (below) below
begin
  definition below_rc_def [simp]:
    "(op ⊑) ≡ (λx y . myBelow x y)"
  instance ..
end
```

Now to instantiate `rc` as a partial order, we must prove that the order relation is reflexive, transitive, and antisymmetric. The proofs for the second two are omitted here.

```
lemma refl_less_rc: "(x::('a::po) rc) ⊑ x"
  apply (induction x)
  apply simp
  apply auto
  done

lemma antisym_less_rc: "(x::('a::po) rc) ⊑ y ⟹ y ⊑ x ⟹ x = y"

lemma trans_less_rc: "(x::('a::po) rc) ⊑ y ⟹ y ⊑ z ⟹ x ⊑ z"
```

With these proven, we instantiate `rc` as a partial order.

```
instantiation rc :: (po)po
begin
  instance
  apply intro_classes
  apply (metis refl_less_rc)
  apply (metis trans_less_rc)
  apply (metis antisym_less_rc)
  done
end
```

Up to this point, we have represented the monad using all binary trees. However, not all binary trees are valid elements of the monad. For an element (M, f) of the RC' monad, the function f has to be Scott continuous, thus monotone. We can verify this by making sure that the values get bigger as we traverse the binary tree.

```
fun rc_order :: "('a::po) rc ⇒ bool" where
  "rc_order (Leaf x) = True" |
  "rc_order (Node x l r) = (
    x ⊑ (getValue l []) ∧
    x ⊑ (getValue r []) ∧
    rc_order l ∧
    rc_order r
  )"
```

Now we can prove that if $x \sqsubseteq y$, then if you traverse both trees in the same direction and pick out a value, then the value you pick from x will be less than or equal to the value you pick from y. Here, `xs` is an arbitrary list of booleans.

```
lemma getValue_order: "
  rc_order x ⟹
  rc_order y ⟹
  x ⊑ y ⟹
  getValue x xs ⊑ getValue y xs
"
```

For the Kleisli extension, there are functions of type `'a ⇒ 'b rc`. But not all functions are valid. We need to make sure that all binary trees in the image have the `rc_order` property defined above. We define this property here.

```
definition ordered :: "('a::po ⇒ 'a rc) ⇒ bool" where
  "ordered f = (∀ x.(rc_order (f x)))"
```

Finally, we of course need our functions to be monotone. We can import the theory `Cont` which defines the monotonicity of functions using the definition `monofun`.

```
definition monofun :: "('a ⇒ 'b) ⇒ bool" where
  "monofun f = (∀x y. x ⊑ y ⟶ f x ⊑ f y)"
```

Before proving that the Kleisli extension is monotone, we first show that its image consists of valid elements.

```
lemma "
  monofun f ⟹
  ordered f ⟹
  rc_order x ⟹
  rc_order (flatMap f x)
"
```

We end by proving the monotonicity of the Kleisli extension.

```
lemma "
  monofun f ⟹
  ordered f ⟹
  rc_order x ⟹
  rc_order y ⟹
  x ⊑ y ⟹
  (flatMap f x) ⊑ (flatMap f y)
"
```

Again, all omitted proofs can be found in the appendix.

5.2 Implementation of rPCF

An implementation of Randomized PCF has been developed using the Standard ML programming language (SML). The parser and interpreter for rPCF are adapted from an implementation

```
e ::=  x | n | true | false | succ | pred | iszero |
       if e then e else e | fn x => e | e e |
       rec x => e | {e | e} | (e) |
       let x = e in e end
```

Table 5.1: BNF grammar for rPCF

```
datatype term = AST_ID of string | AST_NUM of int
              | AST_BOOL of bool | AST_ERROR of string
              | AST_SUCC | AST_PRED | AST_ISZERO
              | AST_FUN of (string * rc_term)
              | AST_REC of (string * rc_term)
and lazy rc_term = LEAF of term
                 | AST_APP of (rc_term * rc_term)
                 | AST_IF of (rc_term * rc_term * rc_term)
                 | NODE of (rc_term * rc_term)
```

Figure 5.2: Terms of rPCF

of PCF by Jon Riecke. Source code for Randomized PCF follows the BNF grammar displayed in Table 5.1.

In the grammar, `x` is an variable and `n` is a natural number. `fn x => e` is $\lambda x.e$ and `rec x => e` is $\mu x.e$. `{e | e}` is how to define a random choice between two expressions. It creates a node, Node (e, e). Every `let` expression must be terminated with the `end` keyword.

An SML program parses the source code to form an abstract syntax tree. Nodes of the tree have type `rc_term`, which are terms that can have random choice. Nodes without random choice are of the form `Leaf term`, where terms have no random choice. `rc_term` and `term` are defined in Figure 5.2. In SML, the keyword `and` is used to make mutually recursive datatypes. `AST_ID of string` represents a variable and contains a string. `AST_ERROR` contains a string that will be displayed if an error occurs. The rest of the terms are self-explanatory.

This abstract syntax tree then gets interpreted according to the semantic rules. First a function `subst:rc_term -> string -> rc_term -> rc_term` is defined to perform term substitution. `subst M x N` represents $M[N/x]$. For example, substitution for function application is defined by:

```
subst (AST_APP(e1,e2)) x t = AST_APP((subst e1 x t),(subst e2 x t))
```

This represents the substitution rule, $(e1\ e2)[t/x] = (e1[t/x])(e2[t/x])$.

Now that substitution is defined, a function `interp:rc_term -> rc_term` is defined that reduces terms based on the small step and big step semantics of rPCF. For example, the reduction

```
fun rc_interp_n t n = if n = 0 then getValue t nil else
    (case (interp t) of
        (LEAF x)       => x
      | (NODE (l,r))   => (case (Random.randRange (0,1) rng) of
            0          => rc_interp_n l (n-1)
          | 1          => rc_interp_n r (n-1))
```

Figure 5.3: Random Traversal of rPCF Trees

of a recursive term is given by:

```
interp(LEAF(AST_REC(x,e))) = interp(subst e x (LEAF(AST_REC (x,e))))
```

This implements the transition rule, $\mu x.M \to M[\mu x.M/x]$.

A conditional term of the form `if e1 then e2 else e3` is reduced using:

```
interp (AST_IF (e1, e2, e3)) = (case (interp e1) of
    (LEAF (AST_BOOL true))  => interp e2
  | (LEAF (AST_BOOL false)) => interp e3
  | (NODE (l, r))     => NODE (tails (interp (AST_IF (l, e2, e3))),
                               heads (interp (AST_IF (r, e2, e3))))
```

Here the boolean `e1` is reduced first. If it is the leaf value `true`, then `e2` is reduced and returned. If it is the leaf value `false`, `e3` is reduced and returned. Finally, if `e1` is a random choice of booleans of the form `NODE (l, r)`, then a node is created. The left subtree is created by reducing the conditional statement with `l` as the boolean value. However, the entire reduced term is not used. The function `tails` is applied to extract the left subtree of the reduced term. This matches the behavior of rPCF's semantics. Similarly, the right subtree is filled by reducing the conditional statement with `r`, and the right subtree is extracted using the `heads` function. The entire definitions for `subst` and `interp` can be found in the attached source code.

Applying `interp` to a term results in a semantic value, which only has nodes and leaves. Execution of a program involves a random traversal of this binary tree, using pseudorandom bits supplied by SML's built-in libraries. To ensure that the traversal terminates, a bound is placed on the number of random bits to use. This traversal is done with the function `rc_interp_n`, defined in Figure 5.3.

One important thing to note is that the creation and interpretation of the abstract syntax tree are done lazily. Since only one branch of random choices will be chosen, it would be very inefficient to evaluate the entire tree. Instead, only the branch corresponding to the random choices

```
let factorial = rec f =>
        fn n => if iszero n then 1 else (* n (f (- n 1)))
in
    factorial 5
end
```

Figure 5.4: The Factorial Function in rPCF

```
let choose = fn n => let help = rec h => fn x => fn y =>
             if iszero x
             then y
             else {h (pred x) (* 2 y) | h (pred x) (succ (* 2 y))}
    in help n 0 end
in
    choose 5
end
```

Figure 5.5: Choosing a Random Integer in rPCF

ever gets evaluated. This lazy evaluation also allows for the creation of infinite trees, which are needed to implement randomized algorithms whose error probability converges to zero.

Programming in rPCF

An example program using this implementation of rPCF is:

```
let plus = rec p =>
    fn x => fn y => if iszero x then y else p (pred x) (succ y)
in
    plus 2 3
end
```

This just defines addition using pred and succ and adds 2 and 3 together. There is no randomness involved in evaluating this program; it should always evaluate to 5. This exact program could be used for an implementation of PCF, without randomness. Basic PCF embeds in rPCF so that any rPCF program without randomness will just look like a PCF program. The above code can be altered to include some randomness as follows:

```
let plus = rec p =>
    fn x => fn y => if iszero x then y else p (pred x) (succ y)
in
    plus 2 {3 | 4}
end
```

```
let eq = fn n => fn m => iszero (- n m)
in

let power2 = fn n =>
    let power = rec p => fn m => fn a =>
        if iszero m then a
                    else (p (pred m) (* 2 a))
        in power n 1
    end
in

let twos = rec t => fn n =>
    if (iszero (% n 2)) then (succ (t (/ n 2)))
                        else 0
in

let powmodhelp = rec p => fn x => fn n => fn m => fn a =>
    if iszero n then a
                else (p x (pred n) m (% (* x a) m))
in

let powmod = fn x => fn n => fn m =>
    powmodhelp x n m 1
in

let sloop = rec s => fn x => fn n => fn i =>
    if (iszero i) then false
        else (if (eq 1 (powmod x 2 n)) then false
            else (if (eq (- n 1) (powmod x 2 n))
                then true
                else (s (powmod x 2 n) n (pred i))))
in
```

Figure 5.6: Implementation of Miller-Rabin in rPCF

Now instead of adding 2 and 3, there is a random choice made between 3 and 4. Evaluations of this program will result in 5 or 6, randomly. The number 2 can also be changed into a random choice.

```
let plus = rec p =>
    fn x => fn y => if iszero x then y else p (pred x) (succ y)
in
    plus {1 | 2} {3 | 4}
end
```

In this case, program evaluation will result in either 4 or 6. Note that 5 is not a possible result. This is due to the nature of the Kleisli extension of the random choice monad.

```
let or = fn a => fn b => if a then true else b
in

let miller = fn n => fn a =>
    (let s = twos (pred n) in
    (let d = / (pred n) (power2 s) in
    (let x = powmod a d n in
        (or (eq x 1) (or (eq x (pred n)) (sloop x n s)))
    end) end) end)
in

let millerrabin = fn p =>
    let help = rec h => fn n => fn x => fn y => fn a =>
        if iszero x then
            (if (miller n (+ 2 a)) then (h n y y 0)
                                   else false)
            else {h n (pred x) y (% (* 2 a) (- n 3)) |
                  h n (pred x) y (% (succ (* 2 a)) (- n 3))}
    in
        help p 16 16 0
    end
in

    millerrabin 91

end end end end end end end end end
```

Figure 5.7: Miller-Rabin Implemented in rPCF (Continued)

Defining addition recursively in terms of succ and pred and then defining multiplication in terms of addition is very inefficient. In order to speed up basic mathematical calculations, the implementation of rPCF was changed to use SML's mathematical operations. The implementation uses prefix notation, so `(+ 2 3)` will add 2 and 3. Addition, subtraction, multiplication, division, and the modulo operator are implemented in this fashion. Now a factorial function can be programmed as shown in Figure 5.4.

The randomness seen so far is just using one random bit. Oftentimes, it is desired to obtain some random natural number. This can be accomplished by using multiple random bits. For n random bits, a choice can be made among 2^n natural numbers. Such a choice function can be programmed as shown in Figure 5.5. The term `choose 5` will randomly return a natural number from 0 to 31.

Now this type of choice function can be used to implement a randomized algorithm on natural numbers, such as the Miller-Rabin algorithm. An implementation of Miller-Rabin is included

in Figures 5.6 and 5.7.

This creates a possibly infinite tree of random choices. A random integer is chosen between 2 and $n-2$ using 16 bits. This random integer is then tested according the Miller-Rabin algorithm. If the test result is composite, then `false` is returned. Miller-Rabin is never sure if a number is prime, so if the test result is prime, then another random choice is made. Thus, for any prime number, every branch in the random choice tree is infinite.

Note that in this implementation of the Miller-Rabin algorithm, there is no limit placed on the number of random bits used. This limit only gets placed when the program is executed. Using $16*m$ bits will run the Miller-Rabin test a maximum of m times. If, after using the specified number of random bits, a leaf is not reached, then the term `BOT` is returned. For Miller-Rabin, `BOT` means "probably true".

Finally, we can combine two Miller-Rabin primality tests using a boolean operation like `or`. Executing a program with the term `or (millerrabin 91) (millerrabin 97)` will use the same random bits to test whether 91 or 97 is prime, matching the behavior of Kleisli extension explained in Section 2.5.

Appendix A

Source Code

A.1 Haskell

```
import System.Random
import Control.Monad
import Control.Concurrent.Async

data RC a = Bottom
          | Leaf [Bool] a
          | Node [Bool] a (RC a) (RC a)
          deriving (Show, Read, Eq)

data Coin = Heads
          | Tails
          deriving (Eq, Show, Enum, Bounded)

instance Random Coin where
  random g = case random g of
                (r,g') | r < 1/2   = (Tails, g')
                       | otherwise = (Heads, g')

unit :: a -> RC a
unit x = Leaf [] x

instance Functor RC where
    fmap f (Leaf xs x) = Leaf xs (f x)
    fmap f (Node xs x y z) = Node xs (f x) (fmap f y) (fmap f z)

getValue :: RC a -> [Bool] -> a
getValue (Leaf _ x) _ = x
getValue (Node _ x _ _) [] = x
getValue (Node _ _ l r) (False:xs) = getValue l xs
getValue (Node _ _ l r) (_:xs) = getValue r xs

getTree :: RC a -> [Bool] -> RC a
getTree (Leaf xs x) _ = (Leaf xs x)
getTree t [] = t
getTree (Node _ _ l r) (False:xs) = getTree l xs
getTree (Node _ _ l r) (_:xs) = getTree r xs

kleisli :: (a -> RC b) -> RC a -> RC b
kleisli f (Leaf xs x) = getTree (f x) xs
kleisli f (Node xs x l r) = Node xs (getValue (f x) xs)
                                    (kleisli f l)
                                    (kleisli f r)

instance Monad RC where
        return x = unit x
        m >>= f = kleisli f m


testF :: Int -> RC Int
```

```
testF x = Node [] x (Leaf [False] (x*2)) (Leaf [True] (x*3))

showRC :: RC a -> String
showRC (Leaf xs x) = "Leaf"
showRC (Node xs x l r) = (showRC l) ++ " | Node | " ++ (showRC r)

toss :: IO Coin
toss = randomIO

choose :: RC a -> Int -> IO a
choose (Leaf _ x) _  = return x
choose (Node _ x _ _) 0 = return x
choose (Node _ x l r) n = do
                    c <- randomIO
                   (if (c==Heads) then (choose r (n-1))
                    else (choose l (n-1)))


tree::RC Int
tree = Node [] 1
           (Node [False] 2
                 (Leaf [False, False] 3)
                 (Leaf [False, True] 4))
           (Node [True] 5
                 (Leaf [True, False] 6)
                 (Leaf [True, True] 7))
tree2::RC Int
tree2 = Node [] 1
            (Node [False] 2
                  (Leaf [False, False] 3)
                  (Leaf [False, True] 4))
            (Node [True] 5
                  (Leaf [True, False] 6)
                  (Leaf [True, True] 7))

add1 = do
        i <- (choose tree 2)
        j <- (choose tree 2)
        return (i+j)


add2 = choose (liftM2 (+) tree tree) 2


add3 = do
        (i,j) <- concurrently (choose tree 2) (choose tree 2)
        return (i+j)
```

A.2 Scala

```
package monad

import scala.util.Random

trait RV[+A] {
  override def toString(): String = {
    def s(t: RV[A], indent: String): String = t match {
      case l: Leaf[A] => indent +
                         l.value.toString
      case n: Node[A] => s(n.right, indent + "    ") +
                         "\n" +
                         indent +
                         n.value.toString +
                         "\n" +
                         s(n.left, indent + "    ")
    }
    s(this, "")
  }

  def getValue(word: List[Int]): A = this match {
    case n: Node[A] => if (word == List()) n.value
    else if (word.head == 0) n.left.getValue(word.tail)
    else n.right.getValue(word.tail)
    case l: Leaf[A] => l.value

  }

  def getTree(word: List[Int]): RV[A] = this match {
    case n: Node[A] => if (word == List()) n
    else if (word.head == 0) n.left.getTree(word.tail)
    else n.right.getTree(word.tail)
    case l: Leaf[A] => l
  }

  private def unit[B] (value: B) = Leaf(value)

  def map[B](f: A => B): RV[B] = this match {
    case l: Leaf[A] => Leaf(f(l.value))
    case n: Node[A] => Node(f(n.value), n.left map f, n.right map f)
  }

  def flatMap[B](f: A => RV[B]): RV[B] = {
    def kleisli(t: RV[A], word: List[Int]): RV[B] = {
      t match {
        case l: Leaf[A] => f(l.value).getTree(word)
        case n: Node[A] => Node(f(n.value).getValue(word),
                                kleisli(n.left, word ++ List(0)),
                                kleisli(n.right, word ++ List(1)))
      }
```

```
    }
    kleisli(this, List())
  }

  private def flatten[B] (tree: RV[RV[B]]): RV[B] = tree flatMap (x => x)

  private val oracle = new Random()

  def choose(): A = this match {
    case l: Leaf[A] => l.value
    case n: Node[A] => {
      if (oracle.nextBoolean) n.right.choose() else n.left.choose()
    }
  }

  def choose(num: Int): A = this match {
    case l: Leaf[A] => l.value
    case n: Node[A] => if (num==0) n.value
                       else if (oracle.nextBoolean) n.right.choose(num-1)
                       else n.left.choose(num-1)
  }
}

case class Node[A](value: A, left: RV[A], right: RV[A]) extends RV[A]

case class Leaf[A](value: A) extends RV[A]
```

A.3 Isabelle

A.3.1 RC.thy

```
theory RC
imports Main
begin

datatype 'a rc =
  Leaf 'a |
  Node 'a "'a rc" "'a rc"

fun getValue :: "'a rc ⇒ bool list ⇒ 'a" where
  "getValue (Leaf x) ys  = x" |
  "getValue (Node x l r) [] = x" |
  "getValue (Node x l r) (False # ys) = getValue l ys" |
  "getValue (Node x l r) (True # ys) = getValue r ys"

fun getTree :: "'a rc ⇒ bool list ⇒ 'a rc" where
  "getTree (Leaf x) ys = (Leaf x)" |
  "getTree r [] = r" |
  "getTree (Node x l r) (False # ys) = getTree l ys" |
  "getTree (Node x l r) (True # ys) = getTree r ys"

lemma [simp]: "getTree t [] = t"
  apply (induct t)
  apply auto
  done

fun fmap :: "('a => 'b) => 'a rc => 'b rc" where
  "fmap f (Leaf x) = Leaf (f x)" |
  "fmap f (Node x l r) = Node (f x) (fmap f l) (fmap f r)"

fun unit :: "'a ⇒ 'a rc" where
  "unit x = Leaf x"

fun kleisli :: "('a ⇒ 'b rc) ⇒
                'a rc ⇒
                bool list ⇒
                'b rc" where
  "kleisli f (Leaf x) word = getTree (f x) word" |
  "kleisli f (Node x l r) word = Node (getValue (f x) word)
                                      (kleisli f l (word @ [False]))
                                      (kleisli f r (word @ [True]))"

fun flatMap :: "('a ⇒ 'b rc) ⇒
                'a rc ⇒
                'b rc" where
  "flatMap f x = kleisli f x []"

fun inTree :: "'a rc ⇒
```

```
                bool list ⇒
                bool" where
  "inTree t [] = True" |
  "inTree (Leaf x) ys = False" |
  "inTree (Node x l r) (False # ys) = inTree l ys" |
  "inTree (Node x l r) (True # ys) = inTree r ys"

lemma [simp]: "getTree t ys = Leaf a ⟹
               getTree t (ys @ b) = Leaf a"
  apply (induct t ys rule: getTree.induct)
  apply simp_all
  done

lemma "flatMap f (unit x) = f x"
  by simp


lemma [simp]: "kleisli unit t ys = t"
  apply (induction t arbitrary: ys)
  apply auto
  done

lemma "(flatMap unit) x = x"
  by auto

lemma [simp]: "getValue (k (getValue t [])) [] =
               getValue (kleisli k t []) []"
  apply (induct t)
  apply auto
  done

lemma "getTree (Node a l r) (b # ys) =
       getTree (getTree (Node a l r) [b]) ys"
  apply (induct b)
  apply auto
  done

lemma tree: "getTree t (xs @ ys) = getTree (getTree t xs) ys"
  apply (induction t xs rule:getTree.induct)
  apply auto
  done

lemma [simp]: "getTree (getTree t xs) ys = getTree t (xs @ ys)"
  by (metis tree)

lemma tree2: "getTree t (x # ys) = getTree (getTree t [x]) ys"
  apply simp
  done

lemma "getValue ((flatMap k) ((flatMap h) t)) [] =
       getValue ((flatMap (λx. (flatMap k) (h x))) t) []"
  apply auto
```

```
  apply (induct t)
  apply auto
  done

lemma list: "xs @ a # ys = (xs @ [a]) @ ys"
  by auto

lemma kleisli: "kleisli k (getTree t ys) (xs @ ys) =
                getTree (kleisli k t xs) ys"
  apply (induct t ys arbitrary: xs rule:getTree.induct )
  apply auto
  apply (metis list)
  apply (metis list)
  done

lemma getValTree: "getValue t (xs @ ys) =
                   getValue (getTree t xs) ys"
  apply (induct t xs rule: getTree.induct)
  apply auto
  done

lemma getValKleisli: "getValue (k (getValue t ys)) (xs @ ys) =
                      getValue (kleisli k t xs) ys"
  apply (induct t ys arbitrary: xs rule:getValue.induct)
  apply auto
  apply (metis getValTree)
  apply (metis append_Cons append_Nil append_assoc)
  apply (metis append_Cons append_Nil append_assoc)
  done

lemma kleisli2: "kleisli k (kleisli h t ys) ys =
                 kleisli (λx. kleisli k (h x) []) t ys"
  apply (induct t arbitrary: ys)
  apply auto
  apply (metis append_Nil kleisli)
  apply (metis append_Nil getValKleisli)
  done

lemma "(flatMap k) ((flatMap h) t) =
       (flatMap (λx. (flatMap k) (h x))) t"
  apply auto
  apply (induction t)
  apply auto
  apply (metis kleisli2)
  apply (metis kleisli2)
  done

end
```

A.3.2 RCOrder.thy

```
theory RCOrder
imports RC Porder Cont Fun_Cpo
begin

fun myBelow :: "'a::below rc ⇒
                'a rc ⇒ bool" where
  "myBelow (Leaf x) (Leaf y) = below x y" |
  "myBelow (Leaf x) (Node y l r) = below x y" |
  "myBelow (Node x l r) (Leaf y) = False" |
  "myBelow (Node x l r) (Node y l2 r2) = (below x y ∧
                                          myBelow l l2 ∧
                                          myBelow r r2)"

instantiation rc :: (below) below
begin
  definition below_rc_def [simp]:
    "(op ⊑) ≡ (λx y . myBelow x y)"
  instance ..
end

lemma refl_less_rc: "(x::('a::po) rc) ⊑ x"
  apply (induction x)
  apply simp
  apply auto
  done

lemma antisym_help: "myBelow (Node x l r) (y::('a::po) rc) ⟹
                     myBelow y (Node x l r) ⟹
                    (Node x l r) = y"
  apply (induction y rule:myBelow.induct)
  apply auto
  apply (metis below_antisym)
  apply (metis below_antisym)
  done

lemma antisym_less_rc: "(x::('a::po) rc) ⊑ y ⟹
                        y ⊑ x ⟹
                        x = y"
  apply (induction x)
  apply auto
  apply (induction y)
  apply auto
  apply (metis below_antisym)
  apply (metis antisym_help)
  done

lemma trans_help: "myBelow (Leaf (y::('a::po))) ya ⟹
                   myBelow ya (Leaf x) ⟹
                   y ⊑ x"
```

```
  apply (induction ya)
  apply auto
  apply (metis below_trans)
  done

lemma trans_help2: "(⋀y. myBelow l2 y ⟹
                         myBelow y l ⟹
                         myBelow l2 l) ⟹
                    myBelow (Node y l2 r2) ya ⟹
                    myBelow ya (Node x l r) ⟹
                    myBelow l2 l"
  apply (induction ya)
  apply auto
  done

lemma trans_help3: "(⋀y. myBelow r2 y ⟹
                         myBelow y r ⟹
                         myBelow r2 r) ⟹
                    myBelow (Node y l2 r2) ya ⟹
                    myBelow ya (Node x l r) ⟹
                    myBelow r2 r"
  apply (induction ya)
  apply auto
  done

lemma trans_less_rc: "(x::('a::po) rc) ⊑ y ⟹
                      y ⊑ z ⟹
                      x ⊑ z"
  apply (induction x arbitrary: y rule:myBelow.induct)
  apply auto
  apply (metis trans_help)
  apply (metis myBelow.simps(3) rc.exhaust)
  apply (metis (full_types) below_trans myBelow.simps(1)
                            myBelow.simps(2)
                            myBelow.simps(4)
                            rc.distinct(1)
                            rc.exhaust)
  apply (metis (full_types) below_refl
                            box_below
                            myBelow.simps(3)
                            myBelow.simps(4)
                            rc.exhaust)
  apply (metis trans_help2)
  apply (metis trans_help3)
  done

instantiation rc :: (po)po
begin
  instance
  apply intro_classes
  apply (metis refl_less_rc)
  apply (metis trans_less_rc)
```

```
  apply (metis antisym_less_rc)
  done
end

fun rc_order :: "('a::po) rc ⇒ bool" where
  "rc_order (Leaf x) = True" |
  "rc_order (Node x l r) = (x ⊑ (getValue l []) ∧
                            x ⊑ (getValue r []) ∧
                            rc_order l ∧
                            rc_order r)"

lemma getValue_order_help: "(⋀xs. getValue l [] ⊑
                                  getValue l xs) ⟹
                            (⋀xs. getValue r [] ⊑
                                  getValue r xs) ⟹
                            x ⊑ getValue l [] ⟹
                            x ⊑ getValue r [] ⟹
                            rc_order l ⟹
                            rc_order r ⟹
                            x ⊑ getValue (Node x l r) ys"
  apply (induction ys)
  apply auto
  apply (metis (full_types) below_refl
                            box_below
                            getValue.simps(3)
                            getValue.simps(4))
  done

lemma getValue_order_help2: "x ⊑ getValue t [] ⟹
                             rc_order t ⟹
                             x ⊑ getValue t ys"
  apply (induction ys rule:getValue.induct)
  apply auto
  apply (metis below_trans)
  apply (metis below_trans)
  done

lemma getValue_order_help3: "rc_order x ⟹
                             getValue x xs ⊑ getValue x (xs @ ys)"
  apply (induction xs arbitrary: ys rule: getValue.induct)
  apply auto
  apply (induction ys)
  apply auto
  apply (metis (full_types) getValue.simps(2)
                            getValue_order_help2
                            po_eq_conv
                            rc_order.simps(2))
  done

lemma getValue_order_help4: "rc_order xa ⟹
                             myBelow xa (Node x l r) ⟹
                             x ⊑ getValue l [] ⟹
```

```
                                x ⊑ getValue r [] ⟹
                                rc_order l ⟹
                                rc_order r ⟹
                                getValue xa [] ⊑ x"
  apply (induction xa)
  apply (metis getValue.simps(1)
               myBelow.simps(2))
  apply (metis getValue.simps(2)
               myBelow.simps(4))
  done

lemma getValue_order_help5: "(⋀x. rc_order x ⟹
                                     myBelow x l ⟹
                                     getValue x ys ⊑
                                     getValue l ys) ⟹
                                rc_order xa ⟹
                                myBelow xa (Node x l r) ⟹
                                x ⊑ getValue l [] ⟹
                                x ⊑ getValue r [] ⟹
                                rc_order l ⟹
                                rc_order r ⟹
                                getValue xa (False # ys) ⊑ getValue l ys"
  apply (induction xa)
  apply auto
  apply (metis below_trans getValue_order_help2)
  done

lemma getValue_order_help6: "(⋀x. rc_order x ⟹
                                     myBelow x r ⟹
                                     getValue x ys ⊑
                                     getValue r ys) ⟹
                                rc_order xa ⟹
                                myBelow xa (Node x l r) ⟹
                                x ⊑ getValue l [] ⟹
                                x ⊑ getValue r [] ⟹
                                rc_order l ⟹
                                rc_order r ⟹
                                getValue xa (True # ys) ⊑ getValue r ys"
  apply (induction xa)
  apply auto
  apply (metis below_trans
               getValue_order_help2)
  done

lemma getValue_order: "rc_order x ⟹
                       rc_order y ⟹
                       myBelow x y ⟹
                       getValue x xs ⊑ getValue y xs"
  apply (induction xs arbitrary: x rule:getValue.induct)
  apply auto
  apply (metis getValue.simps(1)
               myBelow.simps(1)
```

```
                myBelow.simps(3)
                rc_order.cases)
  apply (metis getValue_order_help4)
  apply (metis getValue_order_help5)
  apply (metis getValue_order_help6)
  done

definition ordered :: "('a::po ⇒ 'a rc) ⇒ bool" where
  "ordered f = (∀x.(rc_order (f x)))"

lemma getTree_ordered: "rc_order x ⟹ rc_order (getTree x xs)"
  apply (induction x xs rule:getTree.induct)
  apply auto
  done

lemma getTree_order_help: "x ⊑ getValue t [] ⟹ myBelow (Leaf x) t"
  apply (metis getValue.simps(1)
               getValue.simps(2)
               myBelow.simps(1)
               myBelow.simps(2)
               rc.exhaust)
  done

lemma getTree_order_help2: "rc_order y ⟹
                            myBelow (Leaf x) y ⟹
                            myBelow (Leaf x) (getTree y ys)"
  apply (induction y ys rule:getTree.induct)
  apply auto
  apply (subgoal_tac "myBelow (Leaf x) l")
  apply auto
  apply (metis getTree_order_help
               rev_below_trans)
  apply (subgoal_tac "myBelow (Leaf x) r")
  apply auto
  apply (metis getTree_order_help
               rev_below_trans)
  done


lemma getTree_order: "rc_order x ⟹
                      rc_order y ⟹
                      myBelow x y ⟹
                      getTree x xs ⊑ getTree y xs"
  apply (induction y xs arbitrary: x rule:getTree.induct)
  apply auto
  apply (metis getTree.simps(1)
               myBelow.simps(3)
               rc.exhaust)
  apply (case_tac xa)
  apply auto
  apply (metis getTree.simps(1)
               getTree_order_help
```

```
                rc_order.simps(1)
                rev_below_trans)
  apply (case_tac xa)
  apply auto
  apply (metis getTree.simps(1)
                getTree_order_help
                rc_order.simps(1)
                rev_below_trans)
  done

lemma kleisli_order_help: "(⋀xs. rc_order (kleisli f l xs)) ⟹
                           ∀x y. x ⊑ y ⟶
                                 myBelow (f x) (f y) ⟹
                           ∀x. rc_order (f x) ⟹
                           x ⊑ getValue l [] ⟹
                           rc_order l ⟹
                           getValue (f x) xs ⊑
                           getValue (kleisli f l (xs @ [b])) []"
  apply (induction l)
  apply auto
  apply (metis box_below
                getValTree
                getValue_order
                getValue_order_help3)
  apply (subgoal_tac "f x ⊑ f a")
  apply (subgoal_tac "getValue (f x) xs ⊑ getValue (f a) xs")
  apply (metis below_trans
                getValue_order_help3)
  apply (metis getValue_order)
  apply auto
  done

lemma kleisli_order: "monofun f ⟹
                      ordered f ⟹
                      rc_order x ⟹
                      rc_order (kleisli f x xs)"
  apply (simp only: monofun_def)
  apply (simp only: ordered_def)
  apply auto
  apply (induction x arbitrary: xs rule:rc_order.induct)
  apply auto
  apply (metis getTree_ordered)
  apply (metis kleisli_order_help)
  apply (metis kleisli_order_help)
  done

lemma "monofun f ⟹
       ordered f ⟹
       rc_order x ⟹
       rc_order (flatMap f x)"
  apply (metis flatMap.simps
               kleisli_order)
```

```
  done

lemma order_myBelow: "x ⊑ y ⟹ myBelow x y"
  apply auto
  done

lemma "rc_order (f x) ⟹
       rc_order (f a) ⟹
       myBelow (f x) (f a) ⟹
       myBelow (getTree (f x) ys) (getTree (f a) ys)"
  apply (metis (hide_lams, mono_tags)
               getTree_order
               order_myBelow)
  done

lemma myBelow_alt: "getValue x [] ⊑ y ⟹
                    getTree x [False] ⊑ l ⟹
                    getTree x [True] ⊑ r ⟹
                    myBelow x (Node y l r)"
  apply auto
  apply (induction x)
  apply auto
  done

lemma kleisli_monotone_help: "(⋀x. rc_order x ⟹
                                   myBelow x l ⟹
                                   myBelow (kleisli f x (word @ [False]))
                                           (kleisli f l (word @ [False]))) ⟹
                              (⋀x. rc_order x ⟹
                                   myBelow x r ⟹
                                   myBelow (kleisli f x (word @ [True]))
                                           (kleisli f r (word @ [True]))) ⟹
                              ∀x y. x ⊑ y ⟶
                                   myBelow (f x) (f y) ⟹
                              ∀x. rc_order (f x) ⟹
                              rc_order xa ⟹
                              myBelow xa (Node x l r) ⟹
                              x ⊑ getValue l [] ⟹
                              x ⊑ getValue r [] ⟹
                              rc_order l ⟹
                              rc_order r ⟹
                              myBelow (kleisli f xa word)
                                      (Node (getValue (f x) word)
                                      (kleisli f l (word @ [False]))
                                      (kleisli f r (word @ [True])))"
  apply (induction xa)
  apply auto
  apply (rule myBelow_alt)
  apply auto
  apply (metis getValKleisli
               getValue.simps(1)
               getValue.simps(2)
```

```
                getValue_order
                kleisli.simps(1)
                kleisli.simps(2))
  apply (metis (hide_lams, no_types)
                below_trans
                getTree_order_help
                kleisli.simps(1)
                po_eq_conv
                rc_order.simps(1))
  apply (metis (hide_lams, no_types)
                below_trans
                getTree_order_help
                kleisli.simps(1)
                po_eq_conv
                rc_order.simps(1))
  apply (metis getValue_order)
  done

lemma kleisli_monotone: "∀x y. x ⊑ y ⟶
                              myBelow (f x) (f y) ⟹
                         ∀x. rc_order (f x) ⟹
                         rc_order x ⟹
                         rc_order y ⟹
                         myBelow x y ⟹
                         myBelow (kleisli f x xs) (kleisli f y xs)"
  apply (induction f y xs arbitrary: x rule:kleisli.induct)
  apply auto
  apply (case_tac xa)
  apply auto
  apply (metis getTree_order
                order_myBelow)
  apply (metis kleisli_monotone_help)
  done

lemma "monofun f ⟹
       ordered f ⟹
       rc_order x ⟹
       rc_order y ⟹
       x ⊑ y ⟹
       (flatMap f x) ⊑ (flatMap f y)"
  apply (simp only: monofun_def)
  apply (simp only: ordered_def)
  apply auto
  by (metis kleisli_monotone)
```

A.4 Standard ML

A.4.1 Parser

```
Control.Print.printDepth:= 100;
Control.lazysml := true;
open Lazy;

datatype term = AST_ID of string | AST_NUM of int | AST_BOOL of bool
        | AST_ERROR of string | BOT | NAT_FUN of (int -> int)
        | AST_FUN of (string * rc_term) | AST_REC of (string * rc_term)
        | AST_SUCC | AST_PRED | AST_ISZERO
        | AST_PLUS | AST_MINUS | AST_MULT | AST_DIV | AST_MOD
and lazy rc_term = LEAF of term
                 | AST_APP of (rc_term * rc_term)
                 | AST_IF of (rc_term * rc_term * rc_term)
                 | NODE of (rc_term * rc_term)

datatype token = ID of string | NUM of int
  | IFSYM | THENSYM | ELSESYM | BOTSYM | TRUESYM | FALSESYM
  | SUCCSYM | PREDSYM | ISZEROSYM | FNSYM | RECSYM
  | EQUAL | LPAREN | RPAREN | FNARROW | LETSYM | INSYM | ENDSYM | EOF
  | RC | VERT | LBRACE | RBRACE | PLUS | MINUS | MULT | DIV | MOD

signature PCFLEXER =
sig
    val lexfile : string -> token list
    val lexstr : string -> token list
end

structure PCFlexer: PCFLEXER =
struct
  open TextIO

  fun nexttoken strm =
    case input1 strm of
        NONE   => EOF
      | SOME c =>
      if Char.isSpace c then
        nexttoken strm
      else if Char.isAlpha c then
        let
          fun getid id =
        case lookahead strm of
            NONE   => id
          | SOME d =>
              if Char.isAlpha d orelse Char.isDigit d then
            (input1 strm; getid (id ^ str d))
              else
            id
          val ident = getid (str c)
```

```
  in case ident of
     "if"     => IFSYM
   | "then"   => THENSYM
   | "else"   => ELSESYM
   | "bot"    => BOTSYM
   | "true"   => TRUESYM
   | "false"  => FALSESYM
   | "succ"   => SUCCSYM
   | "pred"   => PREDSYM
   | "iszero" => ISZEROSYM
   | "fn"     => FNSYM
   | "rec"    => RECSYM
   | "let"    => LETSYM
   | "in"     => INSYM
   | "end"    => ENDSYM
   | _        => ID ident
  end
else if Char.isDigit c then
  let
    fun getnum num =
  case lookahead strm of
      NONE   => num
    | SOME d =>
        if Char.isDigit d then
      (input1 strm; getnum (10*num + ord d - ord #"0"))
        else
      num
  in
    NUM (getnum (ord c - ord #"0"))
  end
else
  case c of
      #"=" => (case lookahead strm of
           SOME #">" => (input1 strm; FNARROW)
         | _         => EQUAL)
    | #"(" => LPAREN
    | #")" => RPAREN
    | #"|" => VERT
    | #"{" => LBRACE
    | #"}" => RBRACE
    | #"+" => PLUS
    | #"-" => MINUS
    | #"*" => MULT
    | #"/" => DIV
    | #"%" => MOD
    | #"#" =>
    let fun eatline () =
      case input1 strm of
          NONE       => EOF
        | SOME #"\n" => nexttoken strm
        | SOME _     => eatline ()
    in
```

```
            eatline ()
          end
          | _  => (print ("Skipping illegal character " ^ str c ^ ".\n");
            nexttoken strm)

  fun gettokens strm =
    let
      fun gettokens_aux toks =
        let val tok = nexttoken strm
        in
      if tok = EOF then
        (closeIn strm; rev (EOF::toks))
          else
        gettokens_aux (tok::toks)
        end
    in
      gettokens_aux []
    end

  fun lexstr str = gettokens (openString str)

  fun lexfile file = gettokens (openIn file)

end


signature PCFPARSER =
sig
  val parse : token list -> rc_term
end

structure PCFparser : PCFPARSER =
struct
  fun error msg = print (msg ^ "\n")

  fun parseExp (ID v::tail)       = (LEAF (AST_ID v), tail)
    | parseExp (NUM n::tail)      = (LEAF (AST_NUM n), tail)
    | parseExp (TRUESYM::tail)    = (LEAF (AST_BOOL true), tail)
    | parseExp (FALSESYM::tail)   = (LEAF (AST_BOOL false), tail)
    | parseExp (BOTSYM::tail)     = (LEAF (BOT), tail)
    | parseExp (SUCCSYM::tail)    = (LEAF AST_SUCC, tail)
    | parseExp (PREDSYM::tail)    = (LEAF AST_PRED, tail)
    | parseExp (PLUS::tail)       = (LEAF AST_PLUS, tail)
    | parseExp (MINUS::tail)      = (LEAF AST_MINUS, tail)
    | parseExp (MULT::tail)       = (LEAF AST_MULT, tail)
    | parseExp (DIV::tail)        = (LEAF AST_DIV, tail)
    | parseExp (MOD::tail)        = (LEAF AST_MOD, tail)
    | parseExp (ISZEROSYM::tail) = (LEAF AST_ISZERO, tail)
    | parseExp (IFSYM::tail)      =
    let val (ast1, rest1) = parseExps tail
    in
      if hd rest1 = THENSYM then
```

```
    let val (ast2, rest2) = parseExps (tl rest1)
    in
      if hd rest2 = ELSESYM then
    let val (ast3, rest3) = parseExps (tl rest2)
    in
        (AST_IF(ast1, ast2, ast3), rest3)
    end
      else
    (error "Missing else";
      (LEAF (AST_ERROR "Missing else"), [EOF]))
    end
  else
    (error "Missing then";
      (LEAF (AST_ERROR "Missing then"), [EOF]))
end
| parseExp (FNSYM::ID v::FNARROW::tail) =
    let val (ast, rest) = parseExps tail
in
  (LEAF (AST_FUN(v, ast)), rest)
end
| parseExp (FNSYM::ID v::tail) =
(error ("Missing => after fn " ^ v);
 (LEAF (AST_ERROR ("Missing => after fn " ^ v)), [EOF]))
| parseExp (FNSYM::tail) =
    (error "Missing identifier after fn";
 (LEAF (AST_ERROR "Missing identifier after fn"), [EOF]))
| parseExp (RECSYM::ID v::FNARROW::tail) =
    let val (ast, rest) = parseExps tail
    in
  (LEAF (AST_REC(v, ast)), rest)
end
| parseExp (RECSYM::ID v::tail) =
(error ("Missing => after rec " ^ v);
 (LEAF (AST_ERROR ("Missing => after rec " ^ v)), [EOF]))
| parseExp (RECSYM::tail) =
(error "Missing identifier after rec";
 (LEAF (AST_ERROR "Missing identifier after rec"), [EOF]))
| parseExp (LPAREN::tail) =
    let val (ast, rest) = parseExps tail
in
  if hd rest = RPAREN then
    (ast, tl rest)
  else
    (error "Missing )";
      (LEAF (AST_ERROR "Missing )"), [EOF]))
    end
| parseExp (LETSYM::ID v::EQUAL::tail) =
let val (ast1, rest1) = parseExps tail
in
  if hd rest1 = INSYM then
    let val (ast2, rest2) = parseExps (tl rest1)
    in
```

```
        if hd rest2 = ENDSYM then
          (AST_APP (LEAF (AST_FUN(v, ast2)), ast1), tl rest2)
        else
      (error "Missing end";
        (LEAF (AST_ERROR "Missing end"), [EOF]))
      end
    else
      (error "Missing in";
        (LEAF (AST_ERROR "Missing in"), [EOF]))
  end
  | parseExp (LETSYM::ID v::tail) =
  (error "Missing ="; (LEAF (AST_ERROR "Missing ="), [EOF]))
  | parseExp (LETSYM::tail) =
      (error "Missing identifier after let";
        (LEAF (AST_ERROR "Missing identifier after let"), [EOF]))
  | parseExp (LBRACE::tail) =
    let val (ast1, rest1) = parseExps tail
  in
      if hd rest1 = VERT then
      let val (ast2, rest2) = parseExps (tl rest1)
      in
          if hd rest2 = RBRACE then
          (NODE (ast1, ast2), tl rest2)
          else
          (error "Missing right brace";
            (LEAF (AST_ERROR "Missing right brace"), [EOF]))
      end
      else
      (error "Missing vert";
        (LEAF (AST_ERROR "Missing vert"), [EOF]))
    end
  | parseExp [EOF] =
      (error "Unexpected EOF";
        (LEAF (AST_ERROR "Unexpected EOF"), [EOF]))
  | parseExp _ =
      (error "Bad expression";
        (LEAF (AST_ERROR "Bad expression"), [EOF]))

and parseExps tokens =
  let
    val (ast, rest) = parseExp tokens

    fun startsExp (ID s)  = true
  | startsExp (NUM n) = true
  | startsExp tok     =
      tok = TRUESYM orelse tok = FALSESYM orelse tok = SUCCSYM orelse
      tok = PREDSYM orelse tok = ISZEROSYM orelse tok = IFSYM orelse
      tok = FNSYM orelse tok = RECSYM orelse tok = LPAREN orelse
      tok = LETSYM orelse tok = LBRACE orelse tok = BOTSYM

fun parseExps_aux ast rest =
  if startsExp (hd rest) then
```

```
      let
        val (ast', rest') = parseExp rest
      in
        parseExps_aux (AST_APP (ast, ast')) rest'
      end
    else
      (ast, rest)
    in
      parseExps_aux ast rest
    end

  fun lazy parse tokens =
    let val (ast, rest) = parseExps tokens
    in
      if hd rest = EOF then
    ast
      else
    (error "EOF expected"; LEAF (AST_ERROR "EOF expected"))
    end

end

fun parsefile file = PCFparser.parse (PCFlexer.lexfile file)

fun parsestr str = PCFparser.parse (PCFlexer.lexstr str)
```

A.4.2 Interpreter

```
use "RCParser.sml";

fun getValue (LEAF x) xs = x
|   getValue (NODE (l,r)) nil = BOT
|   getValue (NODE (l,r)) (hd::tl) = if hd then getValue r tl
                                           else getValue l tl
|   getValue v xs = AST_ERROR ("ERROR")

fun getTree (LEAF x) xs = LEAF x
|   getTree (NODE (l,r)) (hd::tail) = if hd then getTree r tail
                                            else getTree l tail
|   getTree x xs = x

fun tails (LEAF x) = LEAF x
|   tails (NODE (l,r)) = l
|   tails x = x

fun heads (LEAF x) = LEAF x
|   heads (NODE (l,r)) = r
|   heads x = x

fun lazy subst (AST_IF (e1, e2, e3)) x t = AST_IF ((subst e1 x t),
                                                   (subst e2 x t),
                                                   (subst e3 x t))
|   subst (AST_APP (e1, e2)) x t = AST_APP ((subst e1 x t),
                                            (subst e2 x t))
|   subst (LEAF (AST_ID v)) x t = if v=x then t else LEAF (AST_ID v)
|   subst (LEAF (AST_FUN (v, e))) x t =
          LEAF (AST_FUN (v, if v=x then e else (subst e x t)))
|   subst (LEAF (AST_REC (v, e))) x t =
          LEAF (AST_REC (v, if v=x then e else (subst e x t)))
|   subst (NODE (l, r)) x t = NODE ((subst l x t), (subst r x t))
|   subst e _ _ = e

fun lazy interp (LEAF (AST_ID x)) = LEAF (AST_ERROR
                                    ("unbound identifier: "^x))
|   interp (LEAF (AST_REC (x, e))) = interp
          (subst e x (LEAF (AST_REC (x, e))))
|   interp (AST_IF (e1, e2, e3)) =
   (case (interp e1) of
      (LEAF (AST_ERROR s))    => LEAF (AST_ERROR s)
    | (LEAF (AST_BOOL true))  => interp e2
    | (LEAF (AST_BOOL false)) => interp e3
    | (NODE (l, r))           => NODE
                                 (tails (interp (AST_IF (l, e2, e3))),
                                  heads (interp (AST_IF (r, e2, e3))))
    | (_)                     => LEAF (AST_ERROR
                                  "if condition must be a bool"))
|   interp (AST_APP (e1, e2)) =
```

```
    (case (interp e1, interp e2) of
       (LEAF (AST_ERROR s), _)           => LEAF (AST_ERROR s)
     | (_, LEAF (AST_ERROR s))           => LEAF (AST_ERROR s)
     | (NODE (l, r), NODE (l2, r2))  => NODE
                                      (tails (interp (AST_APP (l, l2))),
                                       heads (interp (AST_APP (r, r2))))
     | (f, NODE (l, r))          => NODE (tails (interp (AST_APP (f, l))),
                                          heads (interp (AST_APP (f, r))))
     | (NODE (l, r), x)          => NODE (tails (interp (AST_APP (l, x))),
                                          heads (interp (AST_APP (r, x))))
     | (LEAF AST_SUCC, LEAF (AST_NUM n)) => LEAF (AST_NUM (n+1))
     | (LEAF AST_SUCC, _)                => LEAF (AST_ERROR
                                             "succ needs int argument")
     | (LEAF AST_PRED, LEAF (AST_NUM 0)) => LEAF (AST_NUM 0)
     | (LEAF AST_PRED, LEAF (AST_NUM n)) => LEAF (AST_NUM (n-1))
     | (LEAF AST_PRED, _)                => LEAF (AST_ERROR
                                             "pred needs int argument")
     | (LEAF AST_PLUS, LEAF (AST_NUM n)) => LEAF (NAT_FUN (fn m=>m+n))
     | (LEAF AST_PLUS, _)                => LEAF (AST_ERROR
                                             "plus needs int argument")
     | (LEAF AST_MINUS, LEAF (AST_NUM n))=> LEAF (NAT_FUN (fn m=>n-m))
     | (LEAF AST_MINUS, _)               => LEAF (AST_ERROR
                                             "minus needs int argument")
     | (LEAF AST_MULT, LEAF (AST_NUM n)) => LEAF (NAT_FUN (fn m=>n*m))
     | (LEAF AST_MULT, _)                => LEAF (AST_ERROR
                                             "mult needs int argument")
     | (LEAF AST_DIV, LEAF (AST_NUM n))  => LEAF (NAT_FUN (fn m=>n div m))
     | (LEAF AST_DIV, _)                 => LEAF (AST_ERROR
                                             "div needs int argument")
     | (LEAF AST_MOD, LEAF (AST_NUM n))  => LEAF (NAT_FUN (fn m=>n mod m))
     | (LEAF AST_MOD, _)                 => LEAF (AST_ERROR
                                             "mod needs int argument")
     | (LEAF(NAT_FUN f), LEAF(AST_NUM n))=> LEAF (AST_NUM (f n))
     | (LEAF (NAT_FUN f), _)             => LEAF (AST_ERROR
                                             "need int argument")
     | (LEAF AST_ISZERO, LEAF(AST_NUM 0))=> LEAF (AST_BOOL true)
     | (LEAF AST_ISZERO, LEAF(AST_NUM _))=> LEAF (AST_BOOL false)
     | (LEAF AST_ISZERO, _)              => LEAF (AST_ERROR
                                             "iszero needs int argument")
     | (LEAF (AST_FUN (x, e)), v)        => interp (subst e x v)
     | (_, _)                            => LEAF (AST_ERROR
                                             "not a functional application"))
|    interp (NODE (l, r)) = NODE ((interp l), (interp r))
|    interp e = e

val rng = Random.rand(0,1)

fun rc_interp t =
    (case (interp t) of
        (LEAF x)            => LEAF x
      | (NODE (l,r))        => (case (Random.randRange (0,1) rng) of
            0               => rc_interp l
```

```
          | _                => rc_interp r)
      | e                    => e)

fun rc_interp_n t n = if n = 0 then getValue t nil else
    (case (interp t) of
        (LEAF x)             => x
      | (NODE (l,r))         => (case (Random.randRange (0,1) rng) of
            0                => rc_interp_n l (n-1)
          | _                => rc_interp_n r (n-1))
      | e                    => getValue e nil)

fun rc_tree_n t n = if n = 0 then LEAF (getValue (interp t) nil)
            else (case (interp t) of
                   (LEAF x)   => LEAF x
                 | (NODE (l, r)) => NODE (rc_tree_n l (n-1),
                                          rc_tree_n r (n-1))
                 | x => (LEAF (AST_NUM 42)))

fun rc_tree file n = rc_tree_n (parsefile file) n

fun rc file = getValue (rc_interp (parsefile file)) nil

fun rc_n file n = rc_interp_n (parsefile file) n
```

References

[1] E. Moggi, "Notions of computation and monads," *Information and computation*, vol. 93, no. 1, pp. 55–92, 1991.

[2] N. Saheb-Djahromi, "Cpo's of measures for nondeterminism," *Theoretical Computer Science*, vol. 12, no. 1, pp. 19–37, 1980.

[3] A. Jung and R. Tix, "The troublesome probabilistic powerdomain," *Electronic Notes in Theoretical Computer Science*, vol. 13, pp. 70–91, 1998.

[4] D. Varacca, *Probability, nondeterminism and concurrency: two denotational models for probabilistic computation.*

BRICS, 2003.

[5] J. Beck, "Distributive laws," in *Seminar on triples and categorical homology theory*, pp. 119–140, Springer, 1969.

[6] R. Tix, *Continuous D-cones: convexity and powerdomain constructions.*

Shaker, 1999.

[7] R. Tix, K. Keimel, and G. Plotkin, "Semantic domains for combining probability and nondeterminism," *Electronic Notes in Theoretical Computer Science*, vol. 222, pp. 3–99, 2009.

[8] M. Mislove, "Nondeterminism and probabilistic choice: Obeying the laws," in *CONCUR 2000Concurrency Theory*, pp. 350–365, Springer, 2000.

[9] D. Varacca, G. Winskel, *et al.*, "Distributing probability over non-determinism," *Mathematical Structures in Computer Science*, vol. 16, no. 1, pp. 87–113, 2006.

[10] M. Mislove, "Discrete random variables over domains," *Theoretical computer science*, vol. 380, no. 1, pp. 181–198, 2007.

[11] J. Goubault-Larrecq and D. Varacca, "Continuous random variables," in *Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science (LICS'11)*, (Toronto, Canada), pp. 97–106, IEEE Computer Society Press, June 2011.

[12] M. Mislove, "Anatomy of a domain of continuous random variables II," in *Computation, Logic, Games, and Quantum Foundations. The Many Facets of Samson Abramsky*, pp. 225–245, Springer, 2013.

[13] J. Lambek, "From λ-calculus to cartesian closed categories," *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, pp. 375–402, 1980.

[14] D. S. Scott, "Relating theories of the lambda calculus," *To HB Curry: Essays on combinatory logic, lambda calculus and formalism*, pp. 403–450, 1980.

[15] A. Church, "A set of postulates for the foundation of logic," *Annals of mathematics*, pp. 346–366, 1932.

[16] F. Cardone and J. R. Hindley, "History of lambda-calculus and combinatory logic," *Handbook of the History of Logic*, vol. 5, pp. 723–817, 2006.

[17] H. P. Barendregt, *The lambda calculus*, vol. 3.
North-Holland Amsterdam, 1984.

[18] D. Scott, *Continuous lattices*.
Springer, 1972.

[19] G. Gierz, K. H. Hofmann, K. Keimel, J. D. Lawson, M. Mislove, and D. S. Scott, "Continuous lattices and domains, volume 93 of encyclopedia of mathematics and its applications," 2003.

[20] S. Abramsky and A. Jung, "Domain theory," *Handbook of logic in computer science*, vol. 3, pp. 1–168, 1994.

[21] J. Goubault-Larrecq, *Non-Hausdorff Topology and Domain Theory: Selected Topics in Point-Set Topology*, vol. 22.
Cambridge University Press, 2013.

[22] S. Mac Lane, *Categories for the working mathematician*, vol. 5.
springer, 1998.

[23] S. Awodey, *Category theory*.
OUP Oxford, 2010.

[24] S. Abramsky and N. Tzevelekos, "Introduction to categories and categorical logic," in *New structures for physics*, pp. 3–94, Springer, 2011.

[25] A. Jung, *Cartesian closed categories of domains*.
Citeseer, 1989.

[26] P. Wadler, "The essence of functional programming," in *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 1–14, ACM, 1992.

[27] M. W. Mislove, "Topology, domain theory and theoretical computer science," *Topology and its Applications*, vol. 89, no. 1, pp. 3–59, 1998.

[28] K. De Leeuw, E. F. Moore, C. E. Shannon, and N. Shapiro, "Computability by probabilistic machines," *Automata studies*, vol. 34, pp. 183–198, 1956.

[29] J. Gill, "Computational complexity of probabilistic turing machines," *SIAM Journal on Computing*, vol. 6, no. 4, pp. 675–695, 1977.

[30] I. B. M. C. R. Division and M. Rabin, *Probabilistic algorithms*.
1976.

[31] R. Solovay and V. Strassen, "A fast monte-carlo test for primality," *SIAM journal on Computing*, vol. 6, no. 1, pp. 84–85, 1977.

[32] M. O. Rabin, "Probabilistic algorithm for testing primality," *Journal of number theory*, vol. 12, no. 1, pp. 128–138, 1980.

[33] M. Agrawal, N. Kayal, and N. Saxena, "PRIMES is in P," *Annals of mathematics*, pp. 781–793, 2004.

[34] M. Mislove, "Anatomy of a domain of continuous random variables I," *Theoretical Computer Science*, vol. 546, pp. 176–187, 2014.

[35] S. Pettie and V. Ramachandran, "Randomized minimum spanning tree algorithms using exponentially fewer random bits," *ACM Transactions on Algorithms (TALG)*, vol. 4, no. 1, p. 5, 2008.

[36] D. S. Scott, "Stochastic λ-calculi," *Journal of Applied Logic*, vol. 12, no. 3, pp. 369–376, 2014.

[37] U. Berger, J. Blanck, and P. K. Køber, "Domain representations of spaces of compact subsets," *Mathematical Structures in Computer Science*, vol. 20, no. 2, pp. 107–126, 2010.

[38] D. S. Scott, "A type-theoretical alternative to ISWIM, CUCH, OWHY," *Theoretical Computer Science*, vol. 121, no. 1, pp. 411–440, 1993.

[39] C. A. Gunter, *Semantics of programming languages: structures and techniques*.
MIT press, 1992.

[40] S. Abramsky, R. Jagadeesan, and P. Malacaria, "Full abstraction for PCF," *Information and Computation*, vol. 163, no. 2, pp. 409–470, 2000.

[41] J. M. E. Hyland and C.-H. Ong, "On full abstraction for PCF: I, II, and III," *Information and computation*, vol. 163, no. 2, pp. 285–408, 2000.

[42] E. Moggi, *Computational lambda-calculus and monads*.
University of Edinburgh, Department of Computer Science, Laboratory for Foundations of Computer Science, 1988.

Biography

The author was born in Metairie, LA in 1987 and raised in Marrero, LA. From 2005 to 2009, he attended Louisiana State University, graduating with a B.S. in mathematics and a B.S. in computer science. In 2009, the author entered the mathematics Ph.D. program at Tulane University. He earned an M.S. in mathematics in 2011 before finishing the Ph.D. program in May 2016.