Formally Verified Samplers From Discrete Probabilistic Programs

A dissertation presented to

the faculty of

the Russ College of Engineering and Technology of Ohio University

In partial fulfillment

of the requirements for the degree

Doctor of Philosophy

Alexander A. Bagnall

May 2023

This dissertation titled

Formally Verified Samplers From Discrete Probabilistic Programs

by

ALEXANDER A. BAGNALL

has been approved for

the School of Electrical Engineering and Computer Science

and the Russ College of Engineering and Technology by

David W. Juedes, Ph.D.

Professor, School of Electrical Engineering and Computer Science

J. Gordon Stewart, Ph.D.

Maj Mirmirani, Ph.D.

Interim Dean, Russ College of Engineering and Technology

Abstract

BAGNALL, ALEXANDER A., Ph.D., May 2023, Computer Science

Formally Verified Samplers From Discrete Probabilistic Programs (187 pp.)

Directors of Dissertation: David W. Juedes, Ph.D. and J. Gordon Stewart, Ph.D.

This dissertation presents Zar: a formally verified compilation pipeline from discrete probabilistic programs in the conditional probabilistic guarded command language (cpGCL) to proved-correct executable samplers in the random bit model. Zar exploits the key idea that discrete probability distributions can be reduced to unbiased coin-flipping schemes. The compiler pipeline first translates cpGCL programs into *choice-fix* trees, an intermediate representation suitable for reduction of biased probabilistic choices. Choice-fix trees are then translated to coinductive interaction trees for execution within the random bit model. The correctness of the composed translations establishes the *sampling equidistribution theorem*: compiled samplers are correct with respect to the conditional weakest pre-expectation (cwp) semantics of their cpGCL source programs. Zar is implemented and fully verified in the Coq proof assistant. We extract verified samplers to OCaml and Python and empirically validate them on a number of illustrative examples.

We additionally present AlgCo (Algebraic Coinductives), a practical framework for inductive reasoning over coinductive types such as conats, streams, and infinitary trees with finite branching factor, developed during the course of this work to enable convenient formal reasoning for coinductive samplers generated by Zar. The key idea is to exploit the notion of *algebraic CPO* from domain theory to define continuous operations over coinductive types via primitive recursion on "dense" collections of their elements, enabling a convenient strategy for reasoning about algebraic coinductives by straightforward proofs by induction. We implement the AlgCo library in Coq and demonstrate its utility by verifying a stream variant of the sieve of Eratosthenes, a regular expression library based on coinductive tries, and weakest pre-expectation semantics for

potentially nonterminating sampling processes over discrete probability distributions in the random bit model.

Table of Contents

List of Tables

List of Figures

1 Introduction

What is to be done?

Nikolai Chernyshevsky

This dissertation presents a formally verified compiler pipeline from discrete probabilistic programs with unrestricted loops and conditioning to executable samplers in the random bit model that are formally guaranteed to generate samples from the true posterior distribution of their source programs. A novel intermediate representation for probabilistic programs – *choice-fix* trees – enables biased probabilistic choices in source programs to be translated to the random bit model via fair coin-flipping schemes. A novel proof framework called Algebraic Coinductives (AlgCo) based on concepts from domain theory, notably that of *algebraic CPOs*, enables convenient formal reasoning about coinductively encoded samplers.

1.1 Motivation and Purpose

Probabilistic programming has arisen in recent years as a popular tool for probabilistic modeling and Bayesian inference (i.e., *learning from data*). Probabilistic programming languages [WW11, GMR+12, CRN+13, SM16, CGH+17, vdMPYW18, BCJ+19, CTSLM19, SCS+19, SRM21, CMS22] (PPLs) provide a convenient high-level notation for modeling probabilistic processes as programs with random sampling and conditioning on observed data, such that data-flow dependencies between variables encode the causal relationships between them. Among the kinds of models expressible by PPLs are *probabilistic graphical models* (PGMs), widely used in machine learning and statistics with applications in information extraction [PD07], speech recognition [Bil04], medical diagnosis [KN15], computer vision [Isa03], coding theory [RU08], gene/protein modeling [Fri04], social network analysis [FNSB15], and more [KF09].

Unlike conventional programs whose semantics is typically given as sets of possible executions, probabilistic programs (PPs) denote probability distributions over program states. The distribution denoted by a probabilistic program is called its *posterior* distribution, and calculating the posterior of a probabilistic program is called *inference*.

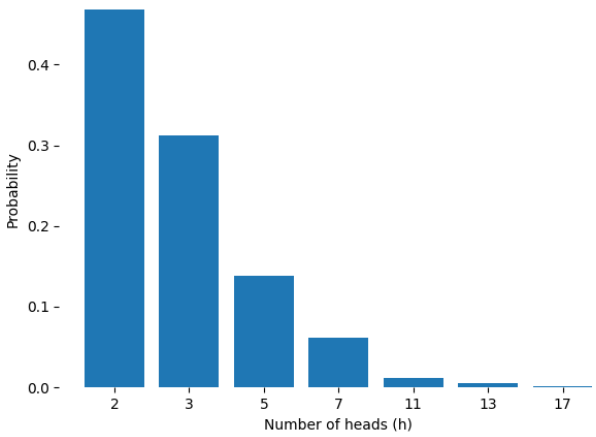
PPLs automate inference by compiling programs to Markov Chain Monte Carlo (MCMC) samplers [Gey11, HTM17, CGH+17] or to other specialized representations [HMB19, HdBM20] for inference. Similarly, systems like Pyro [BCJ+19, WHR21] use techniques for semi-automated inference. Automation helps to separate concerns: the programmer specifies a probabilistic model in a convenient high-level notation, and the inference engine takes care of the details of calculating the posterior distribution. Exact calculation of the posterior, however, is often intractable (exact inference is NP-hard in general [Coo90]), so PPLs typically support sampling from this distribution to enable approximate inference via Monte Carlo methods [RK16].

Unfortunately, bugs in inference engines are especially difficult to detect and diagnose [DLHM18, DZHM19]. Attempts at empirical validation are unlikely to detect small biases and low-probability error conditions. The standard belief propagation algorithm for inference may converge to the wrong solution or fail to converge at all [YFW03], and MCMC samplers may falsely appear to have converged to the desired stationary distribution (known as "pseudo-convergence") [Gey11]. Even the straightforward task of uniform sampling is notoriously susceptible to "modulo bias" [Sec20], leading to violations of cryptographic guarantees [Ngu04, AFG+14, Tho13, BH19, ANT+20] due to improper use of the modulus operator to restrict the range of the uniform distribution. It is therefore difficult to have confidence in the results produced by probabilistic programming systems.

In this dissertation, we present Zar (Bulgarian for "die"): a formally verified compiler from the conditional probabilistic guarded command language

```
1 primes (p : ℚ) :=
2   { b ← true } [p] { b ← false };
3   while b do
4     h ← h + 1;
5     { b ← true } [p] { b ← false }
6   end;
7   observe h is prime
```

(a) 'primes' cpGCL program with geometric posterior over the prime numbers.



(b) True posterior over h with $p = \frac{2}{3}$.

Figure 1.1: Geometric primes program (left) and its posterior distribution over h (right).

(cpGCL [OGJ+18, Kam19]) to proved-correct samplers in the random bit model [VN51, SFRM20b], in which samplers are provided a stream of independent and identically distributed (i.i.d.) random bits drawn from a uniform distribution. Samplers generated by Zar are guaranteed, under reasonable assumptions about the source of randomness (Section 5.3), to produce samples from the true posterior of their source programs, and thus provide a foundation for high-assurance sampling and Monte Carlo-based [RK16] approximate inference. Additionally (Section 8.3), we apply the Zar compiler backend to verify samplers for discrete uniform distributions. The Zar system is implemented and fully verified in the Coq proof assistant.

1.2 Challenges

To understand the challenges, consider the 'primes' cpGCL program in Figure 1.1a, which computes a geometric posterior distribution over the prime numbers as shown in Figure 1.1b. This program combines three fundamental features complicating inference:

1) nonuniform (biased) probabilistic choice, 2) unbounded loop-carried dataflow (a "non-i.i.d." loop [OGJ+18, BSB20]), and 3) conditioning. The variable b is drawn from a Bernoulli distribution with probability p of "heads" (lines 2 and 5). The variable h (with initial value 0 and updated on line 4) counts the number of heads encountered before flipping tails. Finally, the terminal program state is conditioned on h being a prime number (the observe command on line 7).

Eliminating Bias Figure 1.1b shows the posterior of the 'primes' program with bias p specialized to $\frac{2}{3}$. To obtain a sampler in the random bit model, the program must be transformed into a semantically equivalent one in which all choices have bias $\frac{1}{2}$. However, probability expressions in cpGCL can be functions of the program state, so reduction of biased choices is not always possible via direct source-to-source translation (e.g., the probability expression on line 5 could depend on variable h). To address this state dependence and the use of nonuniform biases, we develop a new intermediate representation called choice-fix trees (Section 4.3). We compile cpGCL programs to the choice-fix representation and debias choice-fix trees to generate samplers in the random bit model.

Unbounded and Non-i.i.d. Loops The loop in Figure 1.1a is unbounded; it is not guaranteed to terminate within any fixed number of iterations, and can diverge (though with probability 0) when only heads are flipped. The tasks of sampling and inference are greatly complicated by the infinitary nature of unbounded loops, and thus much previous work on discrete PPs is limited to bounded
loops [CD08, CRN+13, HTM17, HMB19, HdBM20]. Formal reasoning about infinitary computations requires substantial use of *coinduction* [Ber06, KS17, HNDV13], which is notoriously difficult to use in proof assistants like Coq [HNDV13].

Moreover, the loop in Figure 1.1a is "non-i.i.d."; the update of counter variable h on line 4 induces nontrivial data dependence between iterations of the loop, and consequently every value of $h \geq 0$ occurs with nonzero probability (the posterior has *infinite support*). Many interesting probabilistic programs such as the discrete Gaussian (Section 8.4) exhibit such "loop-carried dependence" [AK87]. Prior work on automated inference of unbounded loops and conditioning in probabilistic programs has been restricted to the subclass of i.i.d. loops, i.e., those without loop-carried dependence [BSB20].

Zar compiles the 'primes' program to an executable *interaction tree* (ITree) [XZH+20] formally guaranteed to produce samples from the geometric posterior shown in Fig 1.1b when provided uniform random bits from its environment (see Section 8.2 for empirical evaluation). The coinductive type of ITrees, while suitable for encoding potentially unbounded processes, is deceptively difficult to reason about formally. Coq's built-in mechanism for coinduction is often insufficient [HNDV13, Chl22]. To facilitate reasoning on coinductive representations of samplers, we employ concepts from domain theory such as Scott-continuity [AJ94] and algebraic CPOs [Gun92] (covered in detail in Chapters 6 and 7).

Correctness of Samplers Verified compilers of conventional programming languages like C have somewhat well understood correctness guarantees (though see [PA19]). CompCert [Ler09], for example, uses a simulation argument to prove a form of behavioral equivalence of source and target programs. Writing the specification of a compiler for a PPL is less straightforward. What does "behavioral equivalence" even mean when the result of the compilation pipeline is a probabilistic sampler that depends on a source of randomness?

A key idea of this dissertation is that the proof of correctness of a PPL compiler is essentially a reduction: as input, it takes a source of randomness (in our case, uniformly distributed random bits) and as output it produces a sampler on the posterior distribution

generated by the conditional weakest pre-expectation semantics (cwp) of the program being compiled. We thus reduce the problem of sampling a program's posterior distribution to the comparatively simpler problem of sampling uniformly random bits. Making this reduction work formally means precisely characterizing the input source of randomness and the distributional correctness of the output sampler. We specify the input randomness in Section 5.1, drawing on the classic theory of uniform distribution modulo 1 [Wey16, KN12, BG22]. We characterize distributional correctness by proving that our samplers satisfy an *equidistribution theorem* (Section 5.3) with respect to the cwp semantics of source programs.

Algebraic Coinductives In the course of this work, we found the available options for reasoning about coinductive types (e.g., Coq's built-in cofix tactic and the the well-known paco [HNDV13] library) to be insufficient for the desired goal of defining and reasoning about weakest pre-expectation semantics on coinductive samplers. A technique first developed to overcome these difficulties (see Section 6.1) for the special case of coinductive binary trees has been generalized in domain-theoretic terms to a novel proof framework called *Algebraic Coinductives* (AlgCo), providing a practical system for reasoning about continuous mappings over a wide class of common coinductive types (those which form algebraic CPOs) including conats (Section 6.6), streams (Section 6.7), and infinitary tries (Section 6.8).

1.3 Contributions

This dissertation makes the following contributions:

Concept. We implement Zar: a formally verified compilation pipeline from discrete probabilistic programs with unbounded loops and conditioning to proved-correct executable samplers in the random bit model, exploiting the key idea that discrete distributions can be reduced to unbiased coin-flipping schemes [KY76, SFRM20b],

culminating in the *sampling equidistribution* theorem (Theorem 10) establishing correctness of compiled samplers. The entire system is fully implemented and verified in the Coq proof assistant. To overcome difficulties in coinductive reasoning about compiled samplers in Coq, we develop the novel proof framework AlgCo (Algebraic Coinductives), based on the domain-theoretic notion of algebraic CPO.

Technical. The Zar system includes:

- a formalization of cpGCL and its associated cwp semantics [OGJ+18] (Sections 3.4 and 3.5),
- an intermediate representation for cpGCL programs called *choice-fix* trees (Section 4.1), enabling optimizations and essential program transformations (e.g., elimination of redundant choices and reduction to the random bit model in Section 4.4),
- a compiler pipeline (Section 4.3) from cpGCL to ITree samplers (Section 4.5),
- statement and proof of a general result establishing the correctness of compiled samplers with respect to the cwp semantics of source programs, based on the notion of equidistribution (Chapter 5), and
- a Python 3 package for high-assurance uniform sampling (Section 8.3) as a thin wrapper around proved-correct samplers extracted from Coq.

In addition, we provide an introduction to the basic concepts and proof principles of the AlgCo framework (Chapter 6), including illustrative applications:

- a framework for *lazy coiteration* with conats (Section 6.6),
- a library for reasoning about coinductive streams with application to a formally verified sieve of Eratosthenes (Section 6.7),

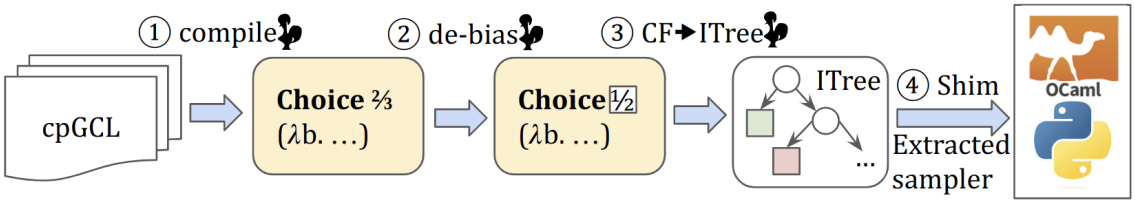


Figure 1.2: Zar pipeline diagram showing (1) the compiler from cpGCL to CF trees (Section 4.1), (2) debiasing of probabilistic choices (Section 4.3), (3) generation of interaction trees from CF trees (Section 4.5), and (4) extraction for efficient execution in OCaml and Python (Chapter 8).

- a verified regular expression library based on coinductive tries (Section 6.8), and
- real-valued weakest pre-expectation semantics on coinductive encodings of sampling processes in the random bit model (Chapter 7).

Evaluation. We perform empirical validation of illustrative examples (Chapter 8) including comparison with FLDR [SFRM20a] and OPTAS [SFRM20b] uniform samplers (Section 8.3) and inference of a posterior over a simulated race between a hare and tortoise (Section 8.5, inspired by the continuous variant in [SK20, Section 1]).

Source Code. Embedded hyperlinks in the PDF point to the underlying Coq sources, hosted publicly on Github in the Zar and AlgCo repositories. At the time of writing, the entire Zar system (including a substantial portion of the AlgCo library) contains 5629 lines of specification and 9339 lines of proof script. The Python 3 package for high-assurance uniform sampling is available in the Zar repository.

Axiomatic Base. We extend the type theory of Coq with excluded middle, constructive indefinite description, and functional extensionality [Cha17, Cha10] (Section 3.1). We also use Coq's axiomatic real number library and extensionality axioms (e.g., Axiom 4 in Section 6.6) for coinductive types (see Section 6.6.1 for discussion).

1.4 Limitations

Zar supports only discrete probabilistic cpGCL programs (which however are naturally suited for many applications [HdBM20]) that terminate either absolutely or almost surely (i.e., with probability 1). Probabilities appearing in cpGCL programs must be rational numbers. We provide no guarantees regarding time/space or entropy usage (number of random bits required to obtain a sample), although we observe near entropy-optimality in some cases (cf. Section 8.3). We verify only the *compiler pipeline*. Verification of cpGCL programs using a program logic that is sound with respect to the cwp semantics is beyond the scope of this work. Proofs on cpGCL programs with respect to their cwp semantics can, however, be composed with our compiler correctness proofs (Theorems 3, 10) to obtain end-to-end guarantees on generated samplers. Directions for future work to overcome these limitations include extending cpGCL to support sampling from a continuous uniform random variate over the unit interval, and compilation to verified Markov chain Monte Carlo (MCMC) sampling processes.

2 Background

> If the language is incorrect ... the people have nowhere to put hand and foot.
>
> *Confucius*

This chapter provides necessary background for understanding the contributions of this dissertation. We begin with a brief overview of probabilistic programming languages (PPLs) (Section 2.1) and inference (Section 2.2). We then introduce our PPL of focus – the conditional probabilistic guarded command language (cpGCL) (Section 2.3) – and its conditional weakest pre-expectation (cwp) semantics (Section 2.4), with extra care given to understanding the semantics of loops (Section 2.5). We give a brief overview of the random bit model (Section 2.7) for random sampling processes and the classic theory of uniform distribution modulo 1 (Section 2.8) upon which the main equidistribution theorem (Theorem 10) is based. We describe the category-theoretic interpretation of the principles of induction and coinduction (Section 2.9) to provide clarity to the issues addressed in Chapter 6. We give a brief account of the interaction tree library of Xia et al. [XZH+20] (Section 2.6) which Zar uses to implement executable sampling semantics of probabilistic programs. Finally, we provide the necessary background on basic order and domain theory (Section 2.10) and measure theory (Section 2.11).

2.1 Probabilistic Programming Languages

Probabilistic programming languages (PPLs) are special-purpose programming languages with built-in support for randomization and probabilistic reasoning. PPLs include standard features from conventional programming languages such as datatypes, variables, conditionals, loops, and recursion, but also provide support for two additional basic constructs:

1. drawing values at random from probability distributions

2. conditioning on observed values of program variables.

PPLs such as Church [GMR+12], Anglican [TvdMYW16], Infer.NET [WW11], IBAL [Pfe01], WebPPL [GS14], Pyro [BCJ+19], and Stan [CGH+17] represent a variety of features and implementation details but all share the same basic notions of drawing random values from a family of built-in probability distributions (e.g., Bernoulli, Uniform, Gaussian, etc.) and conditioning on observations. Programs written in these languages ("probabilistic programs") describe generative models of random processes, encoding joint probability distributions over program variables. In general, the variable types supported may be discrete (e.g., Booleans and integers) or continuous (e.g., real or complex numbers).

2.2 Inference

Probabilistic programs are often used in scientific contexts to perform *Bayesian inference* [McE20]. In the Bayesian paradigm, a probabilistic program is viewed as a model of a real-world data generating process. The model's variables are separated into two classes: *observed* variables (for which samples can be gathered from the real-world process), and *unobserved variables* (hidden variables suspected by the programmer to have causal influence on the observed variables). The model is factored into two components:

1. A *prior* probability distribution over the unobserved variables (often called the *parameters*) representing prior belief about the values they could hold, and

2. a *likelihood function* – a family of distributions ranging over possible values of the parameters, for which the programmer expects some choice of parameters will provide an accurate model of the real-world distribution.

In the absence of any prior knowledge about likely values of the parameters, the prior distribution is typically chosen to maximize entropy (i.e., minimize information) over some class of suitable distributions (the *principle of indifference*). The model is then *conditioned* on a set of particular values for the observed variables, inducing an updated *posterior* distribution over the model's parameters. The act of updating the prior with observations to produce a posterior is called the *Bayesian update* step, and it can be repeated multiple times with each new posterior becoming the prior for the next update step. Inferring the posterior distribution over a model's parameters via Bayesian update constitutes a principled mathematical method for *learning from data*.

A common misconception is that Bayesian inference is defined solely by its use of *Bayes' rule* (explained in the following section) for performing inference. While true that Bayes' rule is the fundamental tool of probability theory for performing the Bayesian update step, the feature that markedly distinguishes the Bayesian paradigm as "Bayesian" is its interpretation of probabilities (e.g., the prior and posterior probabilities on parameter values) as "degrees of certainty" or "belief", in contrast to the frequentist interpretation of probability as the limit of relative frequency over n trials as n goes to $+\infty$.

2.2.1 Bayes' Rule

In this section we derive Bayes' rule from first principles of probability theory and provide an intuitive explanation of the rule.

Recall the definition of the conditional probability of event A given B [PGJ16]:

$$P(A \mid B) = \frac{P(A, B)}{P(B)}. \tag{2.1}$$

Also recall that a joint probability over A and B can be factored via the chain rule of probability into a marginal $P(A)$ and conditional probability $P(B \mid A)$:

$$P(A, B) = P(A)P(B \mid A). \tag{2.2}$$

Rewriting by equation 2.2 in the numerator of the right-hand side of equation 2.1, we obtain Bayes' rule:

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)}.$$

When using Bayes' rule, it is helpful to think of event A as the "hypothesis" and event B as the "evidence." This naming suggests the rule's utility; it is usually easy to calculate the probability $P(B \mid A)$ of the evidence B given a hypothesis A, but more difficult to calculate the probability $P(A \mid B)$ of hypothesis A given evidence B. Bayes' rule allows us to calculate the latter in terms of the former.

When performing Bayesian inference, we begin with some *prior* distribution $P(A)$ over hypotheses A, and we wish to update it in light of new evidence B to obtain a *posterior* distribution $P(A \mid B)$. Using Bayes' rule, we can perform this update in terms of the *likelihood* $P(B \mid A)$ (how well evidence B is explained by hypothesis A) relative to $P(B)$ (i.e., $\int P(B \mid A)\, dA$ – how likely B is over all possible hypotheses).

2.2.2 Exact Inference

Application of Bayes' rule by hand is impractical in most cases, and while it may in principle be automated over a graphical model such as a Bayesian network, the number of terms in the calculation scales exponentially in the number of nodes in the graph, so that approach quickly becomes intractable for large models. Standard algorithms improve upon this but remain exponential in the treewidth of the network [CD08]. Some efforts have been made toward improving performance of exact inference in particular circumstances (see, for example, Holtzen et al. [HMB19] in which exact inference is performed efficiently by weighted model counting from binary decision diagram (BDD) representations of loop-free programs), but state-of-the-art techniques for posterior inference are typically based on methods of approximation [YFW03, Gey11, BCJ+19].

2.2.3 Approximate Inference

When exact inference is infeasible we may resort to methods for approximately estimating the posterior, or more commonly, the expected value of some function of the posterior. Such approximation methods are often based on the idea of generating many samples and computing an empirical estimate from them (as we describe in the following paragraphs), although some (such as the belief propagation algorithm [YFW03]) do not require any random sampling.

Ordinary Monte Carlo The basic framework for sampling-based approximate inference is Ordinary Monte Carlo (OMC), or "i.i.d. Monte Carlo" [BGJM11]. OMC is applicable whenever independent and identically distributed (i.i.d.) samples can be generated from the posterior and is thus quite general because it does not require any special knowledge of the underlying random process.

Suppose that we have a function $g : A \to \mathbb{R}^{\infty}_{\geq 0}$ over sample space A for which we wish to compute the expected value $\mu_g : \mathbb{R}^{\infty}_{\geq 0}$ with respect to a given posterior distribution D over values $a : A$. That is, we wish to compute the following quantity:

$$\mu_g = E_{a \sim D}[g(a)].$$

If exact calculation of μ_g is not possible (due, e.g., to complexity or simply to opaqueness of the model), we may settle to instead calculate the *empirical mean* $\hat{\mu}^n_g$ from n samples $\{x_1, ..., x_n\}$ drawn i.i.d. from D:

$$\hat{\mu}^n_g = \frac{1}{n} \sum_{i=1}^{n} g(x_i).$$

We can also compute the *empirical variance* $\hat{\sigma}^2_n$ using the same set of samples $\{x_1, ..., x_n\}$:

$$\hat{\sigma}^2_n = \frac{1}{n} \sum_{i=1}^{n} (g(x_i) - \hat{\mu}^n_g)^2.$$

From which we can obtain a 95% confidence interval for μ_g of $\hat{\mu}_g^n \pm 1.96 \cdot \frac{\hat{\sigma}_n}{\sqrt{n}}$. Note that OMC sampling follows the square root law; the accuracy of $\hat{\mu}_g^n$ is inversely proportional to the square root of n, and thus every additional significant figure of accuracy requires a hundred-fold increase in the number of samples [Gey11].

The samplers generated from probabilistic programs in this dissertation can be used to perform OMC approximate inference. The equidistribution proved in section 5.3 proves that OMC using a compiled sampler correctly approximates the posterior distribution of the probabilistic program from which it was compiled.

2.3 The Conditional Probabilistic Guarded Command Language

Our object language is the conditional probabilistic guarded command language (cpGCL), introduced by Olmedo et al. [OGJ+18] as an extension of pGCL [MM05] to support conditioning. cpGCL lacks many advanced features of industrial strength PPLs (e.g., the "guide" functions of Pyro [BCJ+19]) but supports the basic operations of probabilistic choice and conditioning on observations, making it an ideal core probabilistic calculus for formalization. The syntax of cpGCL is given by the following grammar:

Figure 2.1: cpGCL syntax

```
C := skip              no-op
   | abort             abort execution
   | x := E            assign value of expression E to variable x
   | observe(G)        condition on predicate G
   | C; C              sequential composition
   | ite(G){C}{C}      conditional branching
   | {C}[p]{C}         probabilistic choice
   | while(G){C}       repetition
```

The cpGCL extends Dijkstra's classic guarded command language [Dij75] with the following constructs:

1. **Probabilistic choice:** Given expression $e : \Sigma \to \mathbb{Q}$ (where Σ is the type of program states) such that $e\,\sigma \in [0, 1]$ for all $\sigma : \Sigma$, command '$\{\ c_1\ \}\ [e]\ \{\ c_2\ \}$' executes command c_1 with probability $e\,\sigma$, or c_2 with probability $1 - e\,\sigma$ in current state σ.

2. **Conditioning:** Given predicate $G : \Sigma \to \mathbb{B}$ on program states, command '**observe** (G)' conditions the posterior distribution of the program on P.

The '**abort**' command is shorthand for a divergent loop, e.g., '**while** $(\lambda\_.\ \mathbf{true})$ {**skip**}'.

2.4 Conditional Weakest Pre-Expectation Semantics

We follow Olmedo et al. [OGJ+18] in interpreting cpGCL programs using conditional weakest pre-expectation (cwp) semantics, a quantitative generalization of weakest precondition semantics [Dij75]. Samplers produced by Zar are proved correct with respect to the cwp semantics of source programs.

Definition 1 (Expectation)**.** *An expectation is a function $f : \Sigma \to \mathbb{R}^{\infty}_{\geq 0}$ mapping program states to nonnegative reals or $+\infty$.*

The cwp semantics interprets programs as expectation transformers: Given a *post-expectation* $f : \Sigma \to \mathbb{R}^{\infty}_{\geq 0}$ and program c : cpGCL, the weakest pre-expectation $\mathsf{wp}\ c\ f : \Sigma \to \mathbb{R}^{\infty}_{\geq 0}$ is a function mapping program states $\sigma : \Sigma$ to the expected value of f over terminal states of c given initial state σ. The name "weakest pre-expectation" comes from being the quantitative analogue of the classic notion of *weakest precondition* in predicate transformer semantics.

Expectations can be seen as a quantitative generalization of predicates in the following sense: given a predicate $Q : \Sigma \to \mathbb{B}$, the weakest pre-expectation of a program C with respect to post-expectation $[Q]$ (Iverson bracket notation – see Table 2.1) maps state

$\sigma : \Sigma$ to the probability that C, when executed from initial state σ, terminates in a final state in which Q holds. Thus, cwp semantics can be used to answer probabilistic queries about the execution behavior of programs. For more background on weakest precondition and its generalization to weakest pre-expectation semantics, see [Kam19, Chapters 2-4].

Table 2.1: Table of notations used in the definition of wp and wlp semantics.

Notation	Definition	
$\mathbf{0}$	$\lambda\sigma.\ 0$	The constant function at 0.
$\mathbf{1}$	$\lambda\sigma.\ 1$	The constant function at 1.
$f[x/E]$	$\lambda\sigma.\ f\ (\sigma[x/E_\sigma])$,	where $\sigma[x/E_\sigma]$ denotes the update of variable x in state σ to the result of evaluating expression E in σ.
$[P]$	$\lambda\sigma.$ if $P\ \sigma$ then 1 else 0	Indicator function for predicate P.
$f + g$	$\lambda\sigma.\ f\ \sigma + g\ \sigma$	Pointwise addition of functions.
$f \cdot g$	$\lambda\sigma.\ f\ \sigma \cdot g\ \sigma$	Pointwise multiplication of functions.
$\frac{f}{g}$	$\lambda\sigma.\ \frac{f\ \sigma}{g\ \sigma}$	Pointwise division of functions.

Table 2.2 gives the definition of wp, as well as a "liberal" variant of wp called the *weakest liberal pre-expectation* (wlp), by induction on the structure of cpGCL programs. Both semantics are extended to support observe commands in anticipation of how they will be used in the conditional extension of wp.

Most of the cases for wp (wlp) are straightforward. Skip leaves the input function f unchanged. Abort yields the function constant at 0 (1) (see Example 1 for explanation). Observe yields the function that checks if predicate G holds in the input state, deferring to f when true and the constant $\mathbf{0}$ when false. Composition of commands corresponds to composition of expectation transformers. Conditionals, in a manner similar to observations, defers to either the expectation transformer of C_1 when G is true in the input

Table 2.2: Weakest pre-expectation (wp) and weakest liberal pre-expectation (wlp) semantics by induction on the syntax of cpGCL.

C	$\mathsf{wp}\ C\ f : \Sigma \to \mathbb{R}^{\infty}_{\geq 0}$	$\mathsf{wlp}\ C\ f : \Sigma \to \mathbb{R}^{\leq 1}_{\geq 0}$
skip	f	f
abort	$\mathbf{0}$	$\mathbf{1}$
$x := E$	$f[x/E]$	$f[x/E]$
observe (G)	$[G] \cdot f$	$[G] \cdot f$
$C_1; C_2$	$\mathsf{wp}\ C_1\ (\mathsf{wp}\ C_2\ f)$	$\mathsf{wlp}\ C_1\ (\mathsf{wlp}\ C_2\ f)$
ite $(G)\ \{C_1\}\ \{C_2\}$	$[G] \cdot \mathsf{wp}\ C_1\ f + [\neg G] \cdot \mathsf{wp}\ C_2\ f$	$[G] \cdot \mathsf{wlp}\ C_1\ f + [\neg G] \cdot \mathsf{wlp}\ C_2\ f$
$\{C_1\}\ [p]\ \{C_2\}$	$p \cdot \mathsf{wp}\ C_1\ f + (1-p) \cdot \mathsf{wp}\ C_2\ f$	$p \cdot \mathsf{wlp}\ C_1\ f + (1-p) \cdot \mathsf{wlp}\ C_2\ f$
while (G) $\{C'\}$	$\mathsf{sup}\ F^n\ \mathbf{0}$, where	$\mathsf{inf}\ F^n\ \mathbf{1}$, where
	$F\ X = [G] \cdot \mathsf{wp}\ C'\ X + [\neg G] \cdot f$	$F\ X = [G] \cdot \mathsf{wlp}\ C'\ X + [\neg G] \cdot f$

state, or to that of C_2 when G is false. Probabilistic choices yield a weighted sum of the two sides, weighted by parameter p. Lastly, the semantics of loops is given as the least (greatest) fixed point of a monotone functional (see more in-depth discussion on the semantics of loops in Section 2.5).

The fundamental difference between wp and wlp is as follows:

• wp encodes *total* program correctness. When posing a query over predicate Q using wp, we are asking "what is the probability that the program terminates *and* does so in a state satisfying Q?". Divergent execution paths (those which never terminate) contribute nothing to the weakest pre-expectation.

• wlp encodes *partial* program correctness. When posing a query over predicate Q using wlp, we are asking "what is the probability that the program either diverges *or*

(a) Without conditioning (b) With conditioning.

Figure 2.2: Diagrams illustrating probability masses of sets of program executions from a fixed initial state, with (right) and without (left) conditioning. The region labeled T represents the mass of executions that terminate in some final state. D is the mass of divergent executions (thus, the mass of $T \cup D$ is equal to 1). $P \subseteq T$ corresponds to the subset of terminating executions that result in a state satisfying predicate P, and $C \subseteq T$ to the subset of executions that are consistent with all observations in the program.

terminates in a state satisfying Q?". Divergent paths contribute their full probability mass to the weakest liberal pre-expectation.

That is, wp and wlp differ primarily in how they deal with nontermination. Furthermore, wlp is defined only on *bounded expectations* $f : \Sigma \to \mathbb{R}^{\leq 1}_{\geq 0}$ (bounded above by 1) as it is only meaningful for probabilities. It follows that wp and wlp coincide for bounded expectations on programs that always terminate (whether absolutely or with probability 1).

To aid in understanding wp and wlp, consider the following scenario. Suppose that we have some program C, initial state σ, and predicate P. Running C from initial state σ terminates with some probability p and diverges the rest of the time with probability $1 - p$. Some portion of the terminating executions result in final states satisfying P. The diagram on the left of Figure 2.2 illustrates this situation, with the area of each region representing the probability mass associated with it. wp C $[P]$ σ can be understood as

computing the area of the region labeled P (relative to the entire space $T \cup D$, a point which is immaterial for now since $T \cup D$ has mass 1), whereas $\mathsf{wlp}\ C\ [P]\ \sigma$ computes the combined area of $P \cup D$.

The conditional weakest pre-expectation (cwp) of post-expectation $f : \Sigma \to \mathbb{R}^{\infty}_{\geq 0}$ with respect to program c : cpGCL is then defined following the approach of [OGJ+18, Kam19]:

Definition 2 (cwp)**.** *For command* C : cpGCL *and expectation* f : *expectation,*

$$\mathsf{cwp}\ C\ f : \Sigma \to \mathbb{R}^{\infty}_{\geq 0} \triangleq \frac{\mathsf{wp}\ C\ f}{\mathsf{wlp}\ C\ \mathbf{1}}.$$

To understand the rationale behind cwp, consider the diagram on the right in Figure 2.2 which generalizes the situation from before to include an additional region C corresponding to program executions with final states satisfying all observation predicates in the program. Now, instead of computing the probability of P relative to *all* executions, we want to compute the probability of ending up within P while also satisfying all observations, relative to *all observation-consistent* executions (which includes divergent executions since they do not contradict any observations). That is, we want to compute $\frac{\mu(P \cap C)}{\mu(C \cup D)}$ (where $\mu(X)$ denotes the probability mass of set X of executions), which is exactly the definition of cwp (the RHS of Definition 2).

2.5 Semantics of Loops

The semantic rules for loops can be derived by noting that we expect loops to be semantically equivalent to their one-step unrollings, which can be expressed (in the case of wp) with the following equation ranging over expectations f:

$$\mathsf{wp}\ (\textbf{while}\ (G)\ \{C\})\ f = \mathsf{wp}\ (\textbf{ite}\ (G)\ \{C;\ \textbf{while}\ (G)\ \{C\}\}\ \{\textbf{skip}\})\ f.$$

We can arrange for this equation to hold by defining the semantics of '**while** (G) $\{C\}$' to be equal to the least fixed point of the functional F given by:

$$\begin{aligned}\mathsf{F}\ X &= \mathsf{wp}\ (\textbf{ite}\ (G)\ \{C;\ X\}\ \{\textbf{skip}\})\ f \\ &= [G] \cdot \mathsf{wp}\ C\ X + [\neg G] \cdot f.\end{aligned}$$

We note that the space of expectations $\Sigma \to \mathbb{R}^{\infty}_{\geq 0}$ is a CPO (Definition 12) with respect to the standard ordering on $\mathbb{R}^{\infty}_{\geq 0}$ lifted pointwise. Thus, we can define the following ω-continuous functional $\mathsf{F} : (\Sigma \to \mathbb{R}^{\infty}_{\geq 0}) \to \Sigma \to \mathbb{R}^{\infty}_{\geq 0}$:

$$\mathsf{F}\ g = [G] \cdot \mathsf{wp}\ C\ g + [\neg G] \cdot f$$

whose continuity follows readily from that of wp which is proven by routine induction (see [Kam19, Theorem 8.7]). Then, by Kleene's fixed-point theorem (e.g., [Gun92, Theorem 4.12]), we have that there exists a least element $\mu\mathsf{F}$ such that:

$$\mu\mathsf{F} = \mathsf{F}\ (\mu\mathsf{F})$$

Or, with the definition of F expanded on the RHS:

$$\mu\mathsf{F} = [G] \cdot \mathsf{wp}\ C\ \mu\mathsf{F} + [\neg G] \cdot f.$$

That is, $\mu\mathsf{F}$ is the least fixed point of F. Most presentations of the wp semantics (e.g., in [Kam19, Table 4.1]) give the semantics of loops in terms of this fixed-point construction. However, as a further corollary of Kleene's theorem, $\mu\mathsf{F}$ can be obtained by taking the supremum of the chain of approximations resulting from iterative application of F starting from the bottom element $\mathbf{0}$.

$$\mu\mathsf{F} = \mathsf{sup}\ (\mathsf{F}^n\ \mathbf{0}).$$

The story is similar for wlp, but we instead iterate F from the top element $\mathbf{1}$ (since the co-domain is now limited to $[0, 1]$) to obtain a descending chain, which corresponds to taking the *greatest* fixed point of F.

Example 1 (Divergent loop)**.** *Consider the weakest pre-expectation of the canonical infinite loop '*$\textbf{while}\ (\lambda\_.\ \textbf{true})\ \{\textbf{skip}\}$*' with respect to post-expectation* f*. Unfolding the first few iterations of the* ω*-chain:*

$$\mathsf{F}^0\ \mathbf{0} = \mathbf{0}$$

$$\mathsf{F}^1\ \mathbf{0} = \mathsf{F}\ (\mathsf{F}^0\ \mathbf{0}) = [\lambda\_.\ \textbf{true}] \cdot \mathsf{wp}\ \textbf{skip}\ \mathbf{0} + [\lambda\_.\ \textbf{false}] \cdot f = \mathbf{0}$$

$$\mathsf{F}^2\ \mathbf{0} = \mathsf{F}\ (\mathsf{F}^1\ \mathbf{0}) = \ldots = \mathbf{0}$$

$$\ldots$$

We quickly see that since the guard condition always evaluates to **true** *the chain remains constant at* **0** *and therefore has supremum equal to* **0***. The situation is similar for the weakest liberal pre-expectation, with the chain being constant at* **1** *and infimum equal to* **1**.

This example illustrates the fact that the **abort** *command is superfluous and could be eliminated without any loss in expressive power, as it is equivalent under both* wp *and* wlp *to* $\textbf{while}\ (\lambda\_.\ \textbf{true})\ \{\textbf{skip}\}$ *or any other loop that diverges with probability* 1.

The wp semantics of loops is not computable in general (and thus cannot be fully automated), but there are some special cases in which it can be computed. Loosely speaking, we say a loop is *i.i.d.* whenever 1) the probability of exiting the loop is the same after every iteration, and 2) there is no data flow across iterations of the loop (i.e., no loop-carried dependence).

Definition 3 (i.i.d. loop)**.** *A loop with guard condition* $G : \Sigma \to \mathbb{B}$ *and body* C : cpGCL *is said to be independent and identically distributed (i.i.d.) with respect to* wp *semantics (and mutatis mutandis for* wlp *semantics) whenever the following condition holds for all states* $\sigma : \Sigma$ *under which* $G_\sigma = \mathit{true}$*, expectations* $f : \Sigma \to \mathbb{R}^\infty_{\geq 0}$*, and indices* $i : \mathbb{N}$*:*

$$\mathsf{wp}\ \mathsf{unroll}_{i+1}\ f\ \sigma = \mathsf{wp}\ C\ [G]\ \sigma \cdot \mathsf{wp}\ \mathsf{unroll}_i\ f\ \sigma + \mathsf{wp}\ C\ ([\neg G] \cdot f)\ \sigma$$

where unroll_n *is the sequence of finite unrollings of the loop defined by:*

$$\mathsf{unroll}_0 = \textbf{ite}\ (G)\ \{\textbf{abort}\}\ \{\textbf{skip}\}$$

$$\mathsf{unroll}_{n+1} = \textbf{ite}\ (G)\ \{C;\ \mathsf{unroll}_n\}\ \{\textbf{skip}\}.$$

We say that a cpGCL program C is i.i.d. whenever every loop in C is i.i.d. with respect to both wp and wlp. Note that this notion of i.i.d.-ness is purely semantic, asserting that the wp semantics of each iteration of the loop is related to the wp semantics of the previous iteration in a regular way. It is possible to characterize the i.i.d. condition conservatively as a syntactic property (see, for example, [Kam19, Definition 5.16]), however, it may reject some loops that are semantically i.i.d.

Example 2 (Non-i.i.d. loop)**.** *To see a loop that is not i.i.d., consider the following program that counts how many times in a row a fair coin flips* **true***:*

Figure 2.3: Loop counter cpGCL program

```
i := 0;
{ x := true} [1/2] { x := false }
while (x = true) {
  i := i + 1;
  { x := true} [1/2] { x := false }
}
```

The probability of exiting the loop on each iteration is constant ($\frac{1}{2}$), but variable i depends on its own value from previous iterations (a loop-carried dependence), and thus the loop (and the program as a whole) is not i.i.d..

The wp semantics of loops in the general case requires calculation of the supremum of an infinite chain of expectations, which in practice amounts to calculation of an infinite

sum, which is clearly not always computable [Chi65]. However, when the loop is i.i.d., the supremum of its corresponding chain *can* be computed via a closed-form solution (namely that of the limit of a geometric series), rendering wp (and wlp and therefore cwp) fully automatable. However, as we can see from the preceding example, the i.i.d. condition totally precludes the use of counters and accumulators, and is therefore an unfortunately draconian restriction. Many interesting probabilistic programs depend on the use of counters and accumulators, so we must be prepared to handle them as well.

The class of i.i.d. programs, however, is not completely vacuous. Consider the following i.i.d. program in which a fair coin is simulated using a biased coin with bias parameter $p \in [0, 1]$. We are interested in the posterior distribution over variable x (expecting it to be that of a fair coin).

Figure 2.4: Simulating a fair coin from one with bias p

```
x := false;
y := false;
while (x = y) {
  { x := true; { y := true } [p] { y := false } }
    [p]
  { x := false; { y := true } [p] { y := false } }
}
```

Or in equivalent pseudo-code:

```
x, y := true, true
while (x = y) {
  x, y ← flip(p), flip(p)
}
```

The probability of terminating the loop is constant at $2p(1 - p)$, and there is no cross-iteration dataflow because x and y are both drawn from a fixed distribution at every

iteration (Bernoulli with parameter p), so it is clearly i.i.d. Now let us consider how we might analyze the posterior distribution over x.

In particular, to answer the query "what is the probability of $x = \textbf{true}$?", it suffices to add up the probabilities of all possible executions that lead to terminal states in which $x = \textbf{true}$. One such execution is obvious: when the first iteration of the loop results in $x = \textbf{true}$ and $y = \textbf{false}$ and so immediately exits with $x = \textbf{true}$. This occurs with probability $p(1 - p)$. Another possibility is that the first iteration ends with $x = y$ (occurring with probability $p^2 + (1 - p)^2$) and then terminates with $x = \textbf{true}$ and $y = \textbf{false}$ in the second iteration, an execution thus having overall probability $(p^2 + (1 - p)^2) \cdot 2p(1 - p)$. Another is that the loop terminates after the third iteration, and another after the fourth iteration, and so on. Clearly, there are infinitely many such execution paths, each corresponding to a different number of iterations of the loop, all with non-zero probability.

Let r denote the probability in any iteration of repeating the loop ($r = p^2 + (1 - p)^2$), and a the probability of exiting the loop in a state satisfying the query condition ($a = p(1 - p)$). Then, the probabilities described above can be written as:

$$a, ra, r^2a, r^3a, r^4a, ...$$

and the total probability (yielding the final answer to our query) is given by $\sum_i ar^i$, a geometric series with first term a and common ratio r, converging to $\frac{a}{1 - r}$ in the limit provided that $|r| < 1$ (which here corresponds to having a non-zero probability of exiting the loop). Given that a and r are both readily computable functions of p, we can easily compute the posterior over x:

$$\frac{a}{1 - r} = \frac{p(1 - p)}{1 - (p^2 + (1 - p)^2)} = \frac{1}{2} \text{ (by algebra)}$$

verifying that the program indeed simulates a fair coin regardless of the bias parameter $p \in [0, 1]$.

The reasoning employed above can be extended to the more general case of computing the pre-expectation of an i.i.d. loop with respect to some post-expectation $f : \Sigma \to \mathbb{R}^{\infty}_{\geq 0}$, leading to the following modified rule for the wp semantics of loops:

$$\mathsf{wp}\ (\textbf{while}\ (G)\ \{C\})\ f = [G] \cdot \frac{\mathsf{wp}\ C\ ([\neg G] \cdot f)}{1 - \mathsf{wp}\ C\ [G]} + [\neg G] \cdot f$$

where wp C $([\neg G] \cdot f)$ (the pre-expectation of f with respect to C, weighted by the probability of exiting the loop) corresponds to the initial term a in the geometric sum and wp C $[G]$ (the probability of repeating the loop) to the common ratio r. In light of this view, i.i.d. loops may just as well be called *geometric* loops. The expression wp C $([\neg G] \cdot f)$ can be understood as the *unnormalized* pre-expectation of C with respect to f, and the expression $1 - \mathsf{wp}\ C\ [G]$ as the *normalization constant* of the loop (constant as it does not depend on f).

For one last example, consider the following card game (from [McE20, Chapter 2]):

Example 3 (Black and white cards)**.** *Suppose you have a deck of three cards. One card is black on both sides, the second is white on both sides, and the third is black on one side and white on the other. Now suppose all three cards are placed into a loose container and shuffled, and one is chosen at random and placed on a table. The side facing up is black. What is the probability that other side is also black?*

We can model the game with the following pseudocode program (where **B** *stands for black and* **W** *for white):*

```
x, y ~ uniform [(B, B), (W, W), (B, W)];
{ front := x; back := y } [1/2] { front := y; back := x };
observe (front = B)
```

*Variables x and y, denoting the two faces of the card, are selected uniformly at random from the three possibilities (***B***,* **B***), (***W***,* **W***), and (***B***,* **W***). We flip a fair coin to determine which of the two sides is facing up, and observe that the front side is* **B***. Since*

the cpGCL *presented here does not directly support the uniform operation (although our formalization in Section 3.4 does), we can implement the uniform choice with an i.i.d. loops as follows:*

```
card_game : cpGCL :=
  x := W; y := B;
  while (x = W ∧ y = B) {
    { x := B} [1/2] { x := W };
    { y := B} [1/2] { y := W }
  };
  { front := x; back := y } [1/2] { front := y; back := x };
  observe (front = B)
```

We can then calculate (using the special rule for i.i.d. loops) the probability that the downward-facing side is also black (where $\sigma_\emptyset$ denotes the empty initial state):

$$\mathsf{Pr}(\mathit{back} = \mathbf{B}) = \mathsf{cwp}\ \mathsf{card\_game}\ [\mathit{back} = \mathbf{B}]\ \sigma_\emptyset = \frac{2}{3}.$$

2.6 Interaction Trees

Interaction trees (ITrees) were introduced by Xia et al. [XZH+20] as a general-purpose coinductive data structure for defining effectful recursive programs that interact with their environment. The ITree library [Xia23] provides a suite of combinators for their construction and a set of formal principles for reasoning about their equivalence (bisimilarity) in Coq. By "effectful" programs, we mean programs that are "impure" in the sense that they may produce side effects such as printing a message or making changes to the program state. An interaction tree computation performs such an effect by raising an event (which may carry data) that is then handled by its environment, possibly providing data in return. The type of interaction trees is equivalent to the following coinductive type in Coq:

```
CoInductive itree (E : Type → Type) (R : Type) : Type ≜
| Ret : R → itree E R
| Tau : itree E R → itree E R
| Vis : ∀A : Type, E A → (A → itree E R) → itree E R.
```

The type itree E R is the type of interaction trees with event family E and return type R. That is, E defines the kinds of events that may be raised (and handled by the environment), and R is the type of the eventual value produced by the computation. The type argument to E determines the type of the environment's response to the event, e.g., an event raised of type $E\ A$ will receive a response value from the environment of type A. An ITree is built using one of three constructors:

- 'Ret x' is a leaf containing value $x : R$.
- 'Tau t' is a "silent step", performing no computation and moving on to subtree t. Divergent computations may be encoded as infinite sequences of 'Tau' constructors.
- 'Vis e f' represents an an interaction with the environment in which event $e : E\ A$ is raised, and the response value $x : A$ from the environment is used to determine the rest of the computation via '$f\ x$'.

Interaction trees can be used to encode potentially non-terminating processes. For example, the following divergent CoFixpoint produces an infinite sequence of Tau nodes:

```
CoFixpoint diverge (E : Type → Type) (R : Type) : itree E R ≜
  Tau (diverge E R).
```

This is an essential feature for us as we require a way to implement samplers corresponding to probabilistic programs which may not terminate along all possible execution paths. Moreover, the ITree library is purely constructive and thus interaction trees built from the combinators it provides can be extracted to OCaml for efficient execution (see Chapter 8). In Section 4.5 we show how to generate interaction tree

representations of discrete probabilistic programs, and in Section 7.6 we define (via the tools of AlgCo developed in Chapter 6) a real-valued expectation semantics on interaction tree samplers corresponding to the weakest pre-expectation semantics of cpGCL.

2.7 The Random Bit Model

In this dissertation we consider a computational model of randomized processes called the *random bit model* [VN51, SFRM20b], within which the basic unit of randomness is the random bit $b \in \{0, 1\}$. A sampler operating in the random bit model may query its environment for a uniformly distributed random bit (i.e., the result of a Bernoulli trial with $p = \frac{1}{2}$) as many times as required to produce its final result. Random bits are generated lazily by the environment on request from the sampling process. All random bits generated by the environment are independent and identically distributed (i.i.d.).

The random bit model provides a number of advantages over alternatives such as the *algebraic* [SFRM20b, Dev86], (sometimes called the *real RAM model* [Blu98]), in which the basic unit of randomness is a random variate U uniformly distributed over the unit interval $[0, 1]$ of real numbers. While analytically convenient, the algebraic model is typically implemented on physical machines via floating point approximations (e.g., the rand function provided by the C standard library and similar systems) which induce sampling error and thus invalidate any theoretical results that hold in the idealized model. The random bit model does not suffer from this deficiency; theoretical results on random bit model samplers continue to hold in practice (under reasonable assumptions about the source of randomness – see Section 5.1).

Moreover, the algebraic model assumes access to infinite entropy (the number of random bits), and is thus insufficient for analysis of the entropy usage of random processes, which is important in scenarios where randomness is at a premium. The random bit model, on the other hand, allows for precise accounting of the number of

random bits consumed by random processes, making it ideal also for analyzing entropy consumption of samplers. The interaction tree samplers compiled from cpGCL programs in Section 4.5 operate within the random bit model where the environment is implemented by a small OCaml shim (Figure 8.1) that lazily generates randomly generated bits in response to **get** events raised by samplers.

2.8 Uniform Distribution Modulo 1

The proofs of correctness for random bit model samplers compiled from probabilistic programs (i.e. the *equidistribution* theorems) in Chapter 5 assume that the samplers are provided access to a sequence of uniformly distributed bit streams. The formal characterization of uniform distribution contained therein borrows heavily from the classic notion of "uniform distribution modulo 1" (u.d.-1) [KN12, BG22]. In this section we provide the necessary background on the theory of u.d.-1. Henceforth we drop the "modulo 1" postfix and consider only real numbers within the interval $[0, 1]$.

Definition 4 (Uniform distribution)**.** *A sequence* $\{x_n\} : \mathbb{N} \to [0, 1]$ *of real numbers in the unit interval is uniformly distributed (u.d.) if for all real endpoints a and b,*

$$\lim_{N\to\infty} \frac{1}{N} \sum_{n=0}^{N-1} [x_n \in [a, b)] = b - a$$

where for $x \in [0, 1]$ *and* $A \subseteq [0, 1]$*,*

$$[x \in A] = \begin{cases} 1 & x \in A \\ 0 & x \notin A \end{cases}$$

A sequence of reals in $[0, 1]$ is u.d. if for all half-open subintervals $[a, b) \subseteq [0, 1]$, the proportion of reals in the sequence falling within $[a, b)$ converges asymptotically to $b - a$ (the length of $[a, b)$). In other words, every half-open subinterval of $[0, 1]$ eventually gets

its "proper share" of reals from $\{x_n\}$. A direct analogue of Definition 4 would suffice as our assumption of randomness if the sets of bitstreams we consider in Section 5.2 always corresponded to half-open intervals. However, since that is not the case, we must generalize from the class of half-open intervals to the larger class Σ_1^0 of *effectively open sets* [BG22].

Definition 5 (Σ_1^0)**.** *A subset $U \subseteq [0,1]$ of the unit interval is effectively open, or Σ_1^0, if it is of the form $U = \bigcup_k I_k$ for some computable sequence $\{I_k\}$ of (possibly empty) open intervals with rational endpoints.*

Definition 6 (Σ_1^0-u.d.)**.** *A sequence $\{x_n\} : \mathbb{N} \to [0,1]$ of real numbers in the unit interval is Σ_1^0-u.d. if for all effectively open sets $U \in \Sigma_1^0$,*

$$\lim_{N\to\infty} \frac{1}{N} \sum_{n=0}^{N-1} [x_n \in U] = \lambda(U)$$

where λ is the Lebesgue measure.

Any concerns that Definition 6 may be vacuous can be quickly put to rest; almost all sequences of reals drawn from $[0,1]$ are Σ_1^0-u.d. [BG22]. Moreover, Σ_1^0-u.d. has deep connections to Martin-Löf randomness [ML66] and Schnorr randomness [DG04] (see Theorem 3 of [BG22]), fortifying our confidence that it is indeed a sensible characterization of uniform randomness.

One further step of generalization to arbitrary measure spaces is required to obtain a form of uniform distribution suitable for the equidistribution theorems of this dissertation:

Definition 7 (μ-u.d.)**.** *Let (A, Σ, μ) be a measure space. Then, a sequence of elements a_n where each $a_n \in A$ is μ-u.d. if for all measurable sets $V \in \Sigma$,*

$$\lim_{N\to\infty} \frac{1}{N} \sum_{n=0}^{N-1} [a_n \in V] = \mu(V).$$

Theorems 9 and 10 are expressed in this generalized form in which the measure μ is induced by the weakest pre-expectation semantics of interaction tree samplers (Definition 124) and cpGCL programs (Definition 2), respectively.

2.9 Induction and Coinduction

This section provides context for understanding the issues addressed by the AlgCo framework in Chapter 6, and can be safely skipped by readers interested only in the high-level picture of the Zar system.

Induction Proof assistants like Coq and Agda allow the programmer to define custom datatypes via inductive definitions. For example, the type natlist of lists of natural numbers can be defined inductively in Coq as follows:

```
Inductive natlist : Type ≜
| nil : natlist
| cons : nat → natlist → natlist.
```

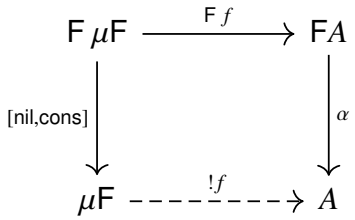
Intuitively, the type natlist can be interpreted semantically as the smallest collection of elements closed under the constructors **nil** and **cons**. I.e., 1) **nil** is a natlist, 2) if l is a natlist then **cons** n l is a natlist for any $n : \mathbb{N}$, and 3) nothing else is a natlist (see [Kle52, Chapter 2] for basic treatment of classical inductive definitions). However, to grasp one of the fundamental difficulties addressed in this dissertation (in Chapter 6), it is helpful for us take a category-theoretic [ML13, Pie91] interpretation of inductive types as *initial* F-*algebras* [Hag89]. We specialize our present discussion to the type natlist, but it is easily generalized to arbitrary inductive definitions.

We begin by noticing that the definition of natlist corresponds to a functor $\mathsf{F} : \mathsf{Type} \to \mathsf{Type}$:

$$\mathsf{F}\, X = 1 + \mathbb{N} \times X$$

where 1 stands for **nil** and $\mathbb{N} \times X$ stands for **cons**).

An F-algebra is a type A (the carrier type) together with an operation $\alpha : F\ A \to A$. The semantic interpretation of natlist is as the initial object in the category of F-algebras (where the objects are F-algebras and the morphisms are F-algebra homomorphisms between them), written μF, with operation $[\mathbf{nil}, \mathbf{cons}] : \mathsf{F}\ \mu\mathsf{F} \to \mu\mathsf{F}$. The universal property of μF as the initial F-algebra is expressed by the following diagram:



The initiality of μF provides a canonical way to construct mappings out of μF into another type A: define an F-algebra operation $\alpha : \mathsf{F}\ A \to A$ and invoke the initiality property to obtain the F-homomorphism $f : \mu\mathsf{F} \to A$. This is the *canonical elimination principle* for the type natlist, and is exposed to the programmer in Coq by the Fixpoint mechanism. For example, we can define the function length : natlist $\to \mathbb{N}$ by choosing carrier type $\mathbb{N}$ and algebra operation $[0, \lambda\_\ n.\ \mathsf{S}\ n]$ and invoking the initiality of natlist (remembering that natlist = μF) to obtain f : natlist $\to \mathbb{N}$ such that $f \circ [\mathbf{nil}, \mathbf{cons}] = [0, \lambda\_\ n.\ \mathsf{S}\ n] \circ \mathsf{F}\ f$. In Coq:

```
Fixpoint length (l : natlist) : nat ≜
  match l with
  | nil ⇒ O
  | cons _ l' ⇒ S (length l')
  end.
```

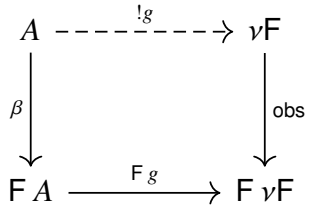
Coinduction The categorical interpretation of inductive types helps us to understand a second class of (less intuitively palatable) user-defined types: *coinductives*, and the fundamental differences between them and inductives. Coinductives are the *categorical dual* of inductives. That is, everything we have said about inductives applies also to coinductives, except with the directions of arrows in the diagrams reversed.

To illustrate, let us define the coinductive type natstream of potentially infinite lists (i.e., *streams*) of natural numbers in Coq (notice that the only meaningful difference from natlist is that natstream is declared as coinductive rather than inductive):

```
CoInductive natstream : Type ≜
| nil : natstream
| cons : nat → natstream → natstream.
```

The intuitive semantic interpretation of natstream is as the *greatest* collection of elements closed under the constructors **nil** and **cons**. The key difference between natlist and natstream is that natlist, being the smallest collection closed under its constructors, contains only finite lists, whereas natstream, being the largest such collection, also contains the infinite lists. Indeed, our desire to compute with infinite lists is the primary reason for passing to the coinductive version of the type. However, the inclusion of infinite elements is not without cost. To understand, let us take a closer look at the categorical interpretation of natstream.

An F-coalgebra is a type A (the seed type) together with an operation $\beta : A \to F\ A$. The semantic interpretation of natstream is as the final object in the category of F-coalgebras (where the objects are F-coalgebras and the morphisms are F-coalgebra homomorphisms between them), written νF, with operation obs : νF $\to$ F νF. The universal property of νF as the final F-coalgebra is expressed by the following diagram:



The finality of νF provides a canonical way to construct mappings *into* νF *from* another type A by defining an F-coalgebra operation $\beta : A \to$ F A and invoking the finality property to obtain the F-homomorphism $g : A \to \nu$F. This is the *canonical introduction principle* for the type natstream, and is exposed to the programmer in Coq by the

CoFixpoint mechanism. For example, we can define the stream of natural numbers increasing from initial seed n by:

```
CoFixpoint nats (n : nat) : natstream ≜
  cons n (nats (S n)).
```

The key thing to notice is that in contrast to natlist, the type natstream *is not equipped with an elimination principle* for defining mappings from streams to other types (nor an induction principle for proving universal statements ranging over all streams). This lack of induction principle for streams and similarly defined coinductive types is sorely felt when attempting to define real-valued semantics of coinductive structures (e.g., weakest pre-expectation style semantics over coinductive samplers as in Section 7.6), and is thus the main driving force behind the theory of algebraic coinductives developed in Chapter 6, in which we carve out a class of coinductives (including streams) that *can* be endowed with a special kind of elimination principle (see Lemma 5) for defining continuous mappings into other types such as $\mathbb{R}^{\infty}_{\geq 0}$ and $\mathbb{P}$.

2.10 Domain Theory

> Divergent series are the devil, and it is a shame to base on them any demonstration whatsoever.
>
> *Niels Henrik Abel*

Here we give the basic definitions of order and domain theory on which the results of this dissertation (including those of algebraic coinductives in Chapter 6) depend.

Definition 8 (Ordered type)**.** *An ordered type (or partial order) is a type A with an order relation $\sqsubseteq_A$ that is reflexive and transitive.*

We typically view $\sqsubseteq_A$ an approximation relation, such that $a \sqsubseteq b$ when a approximates b, that a is somehow a coarser or less informative version of b, or that a

could be further refined in order to obtain b. An ordered type A is said to be *pointed* when it contains a bottom element $\bot_A$ such that $\forall\, a : A, \bot_A \sqsubseteq_A a$. A may also have a top element $\top_A$ such that $\forall\, a : A, a \sqsubseteq_A \top_A$. We often omit the subscripts on $\sqsubseteq_A$ and $\bot_A$ when the ordered type A is clear from context.

Definition 9 (Order equivalence)**.** *Elements x and y of ordered type A are order-equivalent, written $x \simeq y$, when $x \sqsubseteq y$ and $y \sqsubseteq x$.*

Definition 10 (Directed set)**.** *For index type I and ordered type A, a collection $U : I \to A$ is directed when $\forall\, i\ j : I, \exists\, k : I, U\ i \sqsubseteq U\ k \wedge U\ j \sqsubseteq U\ k$, or downward-directed when $\forall\, i\ j : I, \exists\, k : I, U\ k \sqsubseteq U\ i \wedge U\ k \sqsubseteq U\ j$.*

Intuitively, a directed set is one in which all elements are ultimately approximating the same thing. This leads to a natural notion of convergence: a directed set "converges" to its supremum whenever it exists. A special case of directed set is that of ω-chain:

Definition 11 (ω-chain)**.** *For ordered type A, a collection $C : \mathbb{N} \to A$ is an ω-chain when $\forall\, i\ j : \mathbb{N}, i \leq j \Rightarrow C\ i \sqsubseteq C\ j$.*

Definition 12 (Complete partial order)**.** *An ordered type A is a complete partial order (CPO) when every directed $U : \mathbb{N} \to A$ has a supremum, or a lower-complete partial order (lCPO) when every downward-directed $V : \mathbb{N} \to A$ has an infimum. We say that A is a d-lattice when it is both a CPO and lCPO.*

Our choice to specialize to countable directed sets is an artifact of the formalization. We could work with ω-CPOs instead (where every ω-chain has a supremum) but it is generally easier to construct directed sets than ω-chains (e.g., every collection of real numbers is trivially directed).

A few notable d-lattices that appear in this work:

Remark 1 (ℙ is a d-lattice). The type ℙ of propositions ordered by implication (i.e., $P \sqsubseteq Q \iff P \Rightarrow Q$) is a d-lattice with bottom $\bot$ and top $\top$ where for any $U : I \to \mathbb{P}$ (not necessarily (downward-)directed), $\sup(U) \triangleq \exists\, i.U\ i$ and $\inf(U) \triangleq \forall\, i.U\ i$.

Remark 2 ($\mathbb{B}$ is a d-lattice). The type $\mathbb{B}$ of Booleans is a d-lattice with bottom **false** and top **true**.

Remark 3 ($\mathbb{R}^\infty_{\geq 0}$ is a d-lattice). The type $\mathbb{R}^\infty_{\geq 0}$ of nonnegative extended reals is a d-lattice with bottom 0 and top $+\infty$.

Definition 13 (Monotone)**.** *For ordered types A and B, a function $f : A \to B$ is monotone when $\forall\, x\, y : A, x \sqsubseteq y \Rightarrow f\ x \sqsubseteq f\ y$, or antimonotone when $\forall\, x\, y : A, x \sqsubseteq y \Rightarrow f\ y \sqsubseteq f\ x$.*

Definition 14 (Continuous)**.** *For ordered types A and B, a function $f : A \to B$ is continuous when for every directed set $U : \mathbb{N} \to A$, $f\ (\mathsf{sup}\ U) = \mathsf{sup}\ (f \circ U)$, or cocontinuous when $U : \mathbb{N} \to A$, $f\ (\mathsf{sup}\ U) = \mathsf{inf}\ (f \circ U)$.*

We specialize the index type of directed sets to $\mathbb{N}$ for simplicity. Note that (co-)continuity implies (anti)monotonicity.

A continuous function $f : A \to B$ may be viewed as an *approximation transformer*, sending approximate inputs to approximate outputs. That is, if $x_n : A$ is an approximation of $p : A$, then $f\ x_n : B$ is an approximation of $f\ p : B$. The continuity of f can be seen as a guarantee that, if a sequence of outputs of f appears to be converging to some result, then it truly is converging to said result, with no surprises (i.e., discontinuities) anywhere down the line. In other words, it can be trusted that sequences of approximate outputs obtained from sequences of approximate inputs are indeed providing increasingly refined approximations of the true output.

The reader may notice that the dual definition of co-continuity in Definition 14 could instead be obtained through the usual definition of continuity via a reversal of the order relation on B (i.e., swapping to an alternate OType instance). We favor the definition of

co-continuity in terms of suprema and infima instead so that all OType instances can be resolved implicitly, which is only possible when we have a one-to-one mapping from types to instances.

We also sometimes require the following auxiliary notions of continuity:

Definition 15 (Lower-continuous)**.** *For ordered types A and B, a function $f : A \to B$ is lower-continuous (l-continuous) when for every downward-directed set $U : \mathbb{N} \to A$, $f\ (\mathsf{inf}\ U) = \mathsf{inf}\ (f \circ U)$, or l-cocontinuous when $U : \mathbb{N} \to A$, $f\ (\mathsf{inf}\ U) = \mathsf{sup}\ (f \circ U)$.*

Definition 16 (ω-continuous)**.** *Let A and B be partially ordered sets. A function $f : A \to B$ is said to be ω-continuous when for any ω-chain C for which the supremum exists,*

$$f\ (\mathsf{sup}\ C) \simeq_B \mathsf{sup}\ (f \circ U).$$

2.11 Measure Theory

In Chapter 5 we employ concepts from measure theory to state and prove the main equidistribution theorem (Theorem 10) for samplers generated by Zar. This chapter introduces the basic definitions of measure theory necessary for understanding the equidistribution theorem. For more detail we refer the reader to standard texts on measure theory such as [BR07] or [Hal13].

The notion of "measure" is a generalization of the concept of *length* or *size*. For example, the measure of a real interval $[a, b]$ is (under the standard Lebesgue measure) equal to its length $b - a$. Similarly, the measure of a finite collection of natural numbers may be equal (under the appropriate choice of measure) to its cardinality. In general, the concept of measure is rather flexible; for any given space A there can be many different schemes for assigning measures to its subsets. However, to define a measure on A, it is first necessary to specify which of its subsets are considered to be "measurable".

Definition 17 (σ-algebra)**.** *A σ-algebra on a set A is a collection $\Sigma \subseteq \mathcal{P}(A)$ of subsets of A that includes A itself and is closed under countable unions and complements.*

$A \in \Sigma$	(includes A)
$\bigcup_{i=0}^{\infty} U_i \in \Sigma$ when each $U_i \in \Sigma$	(closure under countable union)
$A \setminus U \in \Sigma$ when $U \in \Sigma$	(closure under complement)

An element $U \in \Sigma$ is called a measurable set in A. The power set $\mathcal{P}(A)$ trivially forms a σ-algebra, called the discrete σ-algebra on A.

Definition 18 (Measurable Space)**.** *A measurable space (A, Σ) is a set A equipped with a σ-algebra $\Sigma \subseteq \mathcal{P}(A)$.*

Given a measurable space (A, Σ), a measure is a function assigning to each element of Σ (the measurable subsets of A) either a nonnegative real number or $+\infty$.

Definition 19 (Measure)**.** *Let (A, Σ) be a measurable space. A function $\mu : \Sigma \to \mathbb{R}_{\geq 0}^{\infty}$ is a measure if it satisfies the following properties:*

$0 \leq \mu(U)$ for all $U \in \Sigma$	(non-negativity)
$\mu(\emptyset) = 0$	(null empty set)
$\mu(\bigcup_{i=0}^{\infty} U_i) = \sum_{i=0}^{\infty} \mu(U_i))$, where $\{U_n\}$ is pairwise disjoint and $U_i \in \Sigma$ for all $i : \mathbb{N}$.	(countable additivity)

Definition 20 (Measure Space)**.** *A measure space (A, Σ, μ) is a measurable space (A, Σ) equipped with measure $\mu : \Sigma \to \mathbb{R}_{\geq 0}^{\infty}$.*

We are especially interested in measures that assign probabilities to subsets of a sample space.

Definition 21 (Probability Measure)**.** *A probability measure on measurable space (A, Σ) is a measure $\mu : \Sigma \to \mathbb{R}_{\geq 0}^{\infty}$ such that $\mu(A) = 1$.*

Definition 22 (Probability Space)**.** *A probability space is a measure space* (A, Σ, μ) *such that* μ *is a probability measure.*

The notion is measure is lifted to functions as follows:

Definition 23 (Measurable Function)**.** *Let* (A, Σ_A) *and* (B, Σ_B) *be measurable spaces. A function* $f : A \to B$ *is said to be measurable if the preimage under* f *of measurable sets in* B *are measurable in* A*, i.e.,* $f^{-1}(V) \in \Sigma_A$ *for all* $V \in \Sigma_B$*.*

It is possible to use a measurable function $f : A \to B$ from measure space (A, Σ_A, μ_A) to measurable space (B, Σ_B) to define a measure on B:

Definition 24 (Pushforward Measure)**.** *Let* (A, Σ_A, μ_A) *be a probability space,* (B, Σ_B) *a measurable space, and* $f : A \to B$ *a measurable function from* A *to* B*. The pushforward measure* $f_*(\mu) : \Sigma_B \to \mathbb{R}^{\infty}_{\geq 0}$ *is a measure induced on* B *by letting the measure of* $V \in B$ *be equal to the measure of its preimage in* A*:* $f_*(\mu)(V) = \mu(f^{-1}(V))$*.*

In Section 5.2 we interpret samplers as measurable functions on a standard probability space and use them as in Definition 24 to induce probability measures on the sample space.

3 Formal Foundation

S'il n'existait pas Dieu, il faudrait
l'inventer.

Voltaire

The compiler system described in this dissertation is fully implemented and verified in the Coq proof assistant. Coq exploits a deep connection between functional programming and proof theory (commonly referred to as the "Curry-Howard correspondence" [Cur34, How80, BC85, PCG+10]) to unify programming and proving within a single language. The theoretical results proved in this dissertation are thus "live" [Pie16] in the sense that they apply directly to the executable code of the compiler system rather than to a model that we merely hope to faithfully implement.

In this chapter we present an extension of Coq's type theory with axioms for classical reasoning (Section 3.1) and demonstrate how the axioms can be used to define nonconstructive operators for taking suprema and infima of collections of elements of CPOs (Section 3.2). We show how to extend the wp and wlp expectation transformers to restore a lost symmetry in the presence of conditioning (Section 3.3), and develop formalizations of the cpGCL (Section 3.4) and the extended wp and wlp semantics (Section 3.5).

3.1 Axiomatic Extensions

The underlying logic of Coq is a dependent type theory (an extension of the Calculus of Inductive Constructions [CH88, INR23]) based on Martin-Löf's constructive type theory [Mar84], whose influence can be traced to the intuitionist program initiated by Brouwer [Bro13]. Constructive type theory is notable for its omission of several features usually taken for granted by mathematicians (often working tacitly within the framework of ZFC set theory). The most notorious is the law of excluded middle [Chu28], a subject

of bitter dispute between Brouwer and advocates of the opposing school of "formalism" represented by David Hilbert [Kle52].

Although the type theory of Coq does not include the law of excluded middle by default, it may safely be added as an axiom, so long as one is careful not to introduce further axioms that are incompatible with it. Therefore, with all due respect to intuitionists, we extend the base type theory of Coq with a handful of axioms carefully chosen to enable a classical style of reasoning while preserving consistency of its logic (see [Cha23] for a well-known library that also takes the following three axioms).

Axiom 1 (Function Extensionality)**.** *Let A and B be types, and let $f : A \to B$ and $g : A \to B$ be functions from A to B. Then,*

$$(\forall x : A, f\ x = g\ x) \Rightarrow f = g.$$

Axiom 2 (Excluded Middle)**.** *Let P be a proposition. Then,*

$$P \vee \neg P.$$

Axiom 3 (Constructive Indefinite Description)**.** *Let A be a type and let $P : A \to \mathbb{P}$ be a predicate on A such that there provably exists an $x : A$ such that $P\ x$. Then, we may introduce a term of type $\{x : A \mid P\ x\}$ into the computational universe* Set.

Axiom 3 is more technical in nature than the other two as it pertains to peculiarities of the design of Coq. Coq's type theory is stratified into a hierarchy of "universes" with the two universes Prop and Set sitting parallel at the bottom. Prop is the universe of logical propositions; inhabitants of Prop are considered to be irrelevant for computation (the principle of *proof irrelevance* [GCST19]) and are thus ignored by Coq's extraction mechanism. Inhabitants of Set, on the other hand, are computationally relevant. In short, Set represents the programming fragment of Coq's type theory, and Prop the purely logical fragment.

A clean separation is usually enforced between Prop and Set. One major benefit of this separation is that non-constructive axioms can be safely added to Prop without risk of interfering with the computability of terms in Set. However, the axiom of constructive indefinite description (Axiom 3) allows us to break this separation by transforming proofs of the form $\exists x.\ P\ x$ in Prop into equivalent terms of type $\{x \mid P\ x\}$ (i.e., Σ, or *dependent pair* types) in Set. The benefits of doing this become clear in the next section where we use Axiom 3 to implement convenient operators for expressing suprema and infima in a classical style. The cost, however, is that we lose the guarantee that all inhabitants of Set are computable, and must consequently take extra care not to attempt to compute with terms introduced via Axiom 3 (see Section 6.7.5 for discussion of special cases in which such terms can be implemented by extraction primitives for external execution in OCaml or Haskell), as such terms would cause computation to become blocked [Ler15].

In addition to the axioms listed above, we make use of Coq's standard real numbers library [coq23b] which is built on a non-constructive axiomatization of the reals. We also carefully introduce extensionality axioms for convenient reasoning about equality of coinductive structures (Axioms 4, 5, 6, and 7) (see Section 6.6.1 for discussion).

3.2 Computing Suprema

Many traditional mathematical definitions involve "taking the supremum" of a collection of elements via a sup operator [R+76]. For example, the square root of 2 can be given by the expression $\mathsf{sup}\ \{x : \mathbb{R} \mid x^2 < 2\}$. Unfortunately, since suprema of (possibly infinite) collections are not computable in general, it is not possible to define such an operator within the purely constructive base type theory of Coq. We may however, through daring use of non-constructive axioms, *pretend* that suprema are computable, so that our definitions can be naturally expressed as they appear in traditional presentations as in, e.g., the wp semantics of cpGCL (Definition 28) which defines the semantics of

loops as the supremum of an ascending chain of expectations. In this section we show how to use the axioms of excluded middle (Axiom 2) and constructive indefinite description (Axiom 3) to define convenient operators for taking suprema and infima of countably infinite collections.

We begin with the following somewhat peculiar lemma ex_ex_supremum which says that for any ordered type A and countable collection $f : \mathbb{N} \to A$, there exists an $x : A$ such that *if a supremum exists for* f, x is the supremum of f (and we say "the" supremum because suprema are unique up to order-equivalence by definition).

```
Lemma ex_ex_supremum {A} `{OType A} (f : nat → A) :
  ∃x : A, (∃ b : A, supremum b f) → supremum x f.
Proof. (* omitted *) Qed.
```

The proof of ex_ex_supremum depends on excluded middle (Axiom 2). From it and constructive indefinite description (Axiom 3) we derive the following operator sup_prim which for any ordered type A and countable collection $f : \mathbb{N} \to A$ produces an element $a : A$ such that a is the supremum of f whenever such a supremum provably exists.

```
Definition sup_prim {A} `{OType A} (f : nat → A)
  : { a : A | (∃ b, supremum b f) → supremum a f } ≜
  constructive_indefinite_description _ (ex_ex_supremum _).
```

The sup_prim operator transports ex_ex_supremum from the universe Prop of propositions into the universe Set of computations, allowing for a functional mapping from collections $f : \mathbb{N} \to A$ to their suprema (conditional on the existence of such suprema) within computational contexts. This is an important step toward enabling succinct expression of suprema in a classical style, but the definition is still bogged down by an awkward condition on the existence of suprema. We remedy this by defining the final sup operator as the left projection of sup_prim, and then proving that whenever A is a CPO (meaning that suprema of collections always exist), sup indeed produces the supremum of its input.

```
(* Take the supremum of countable collection 'f'. *)
Definition sup {A} `{OType A} (f : nat → A) : A ≜ proj1_sig (sup_prim f).

(* 'sup f' is the supremum of 'f' whenever 'A' is a CPO and 'f' is directed. *)
Lemma sup_spec {A} `{CPO A} (f : nat → A) :
  directed f → supremum (sup f) f.
Proof. (* omitted *) Qed.
```

Indefinite description (Axiom 3) allows the use of sup even within computational contexts (e.g., the universe Set) (hence the notion of "pretending" that suprema are computable) while preserving the consistency of the logic of Coq. However, computation would become *blocked* [Ler15] if we were to actually attempt to compute with any terms containing applications of sup. Therefore, we must think of any computations expressed using sup or other similar non-constructive operators as being merely part of the correctness specification of the computations that *are* intended for execution, and take care to enforce a clear separation between the computable and noncomputable terms.

The steps are reproduced for infima, yielding an inf operator analogous to sup:

```
Lemma ex_ex_infimum {A} `{OType A} (f : nat → A) :
  ∃x : A, (∃ b : A, infimum b f) → infimum x f.
Proof. (* omitted *) Qed.

Definition inf_prim {A} `{OType A} (f : nat → A) : { a : A | (∃ b, infimum b f) → infimum a f } ≜
  constructive_indefinite_description _ (ex_ex_infimum _).

(* Take the infimum of countable collection 'f'. *)
Definition inf {A} `{OType A} (f : nat → A) : A ≜ proj1_sig (inf_prim f).

(* 'inf f' is the infimum of 'f' whenever 'A' is an lCPO and 'f' is downward−directed. *)
Lemma inf_spec {A} `{lCPO A} (f : nat → A) :
  downward_directed f → infimum (inf f) f.
Proof. (* omitted *) Qed.
```

The supremum and infimum operators described above can be understood as specialized forms of Hilbert's classical epsilon operator [coq23a]). See [Cha10] for

discussion on a similar technique for defining nonconstructive fixpoint operators via Hilbert's epsilon, and the TLC library [Cha23] (an alternative to Coq's standard library for classical reasoning) built around it.

The classic Kleene fixed-point theorem (see, e.g., [Gun92, Theorem 4.12]) is formalized by the following 'iter' construction, taking the least fixed point of ω-continuous automorphism $F : A \to A$ starting from zero element $z : A$. The analogous 'dec_iter' takes the greatest fixed point of decreasing ω-continuous functional F, where decreasing ω-continuous means to preserve infima of decreasing ω-chains. The lemma 'iter_unfold' demonstrates that the term 'iter F z' is indeed a fixed point (up to order equivalence) of F whenever F is ω-continuous and $z \sqsubseteq F\ z$.

```
(* Compute the least fixed point of F by taking the supremum of the
   chain obtained by repeated iteration of F starting from z. *)
Definition iter {A} `{OType A} (F : A → A) (z : A) : A ≜
  sup (iter_n F z).

(* 'iter F z' is a fixed point of 'F'. *)
Lemma iter_unfold {A} `{CPO A} (F : A → A) (z : A) :
  wcontinuous F →
  z ⊑ F z →
  iter F z === F (iter F z).
Proof. (* omitted *) Qed.
```

'iter' is used to implement a cotree analogue of the 'ITree.iter' iteration combinator of the interaction tree library (see Section 7.6) which is essential for compilation of loops.

3.3 Conditional Symmetry

In the non-conditional setting (that is, cpGCL without the 'observe' command), the wp and wlp expectation transformers are known to satisfy a number of basic properties and healthiness conditions such as linearity, monotonicity, and continuity (see [Kam19, Section 4.2]). Among these properties can be seen an elegant symmetry between wp and

wlp for all C : cpGCL:

$$\mathsf{wp}\ C\ \mathbf{0} = \mathbf{0} \tag{3.1}$$

$$\mathsf{wlp}\ C\ \mathbf{1} = \mathbf{1} \tag{3.2}$$

$$\mathsf{wp}\ C\ f + \mathsf{wlp}\ C\ (\mathbf{1} - f) = \mathbf{1}, \text{ where } f \sqsubseteq \mathbf{1}. \tag{3.3}$$

Equation 3.1 says that wp is *strict*; wp C sends $\mathbf{0}$ to $\mathbf{0}$ for any C. Likewise, equation 3.2 says that wlp is *co-strict*, with wlp C sending $\mathbf{1}$ to $\mathbf{1}$ for any C. Both properties are essential for reasoning about their respective expectation transformers. Equation 3.3, however, having many useful corollaries, is perhaps even more valuable. We refer to equation 3.3 as the *invariant sum property*.

When support for conditioning on observations is introduced into the language, this symmetry is lost. wp retains its strictness, but equations 3.2 and 3.3 no longer hold. To see why equation 3.2 fails, note that wlp takes the value $\mathbf{0}$ on observation failure, so it must be the case that wlp $C\ \mathbf{1} < \mathbf{1}$ for any program C with nonzero probability of observation failure. Similarly, equation 3.3 fails because neither wp nor wlp are able to account for the probability mass of observation failure.

These remarks lead us directly to a solution. The desired symmetry can be regained by augmenting wp and wlp with an additional Boolean parameter indicating whether or not to include the probability mass of observation failure, as shown in table 3.1.

Table 3.1: wp and wlp semantics with optional inclusion of probability mass of observation failure.

C	$\mathsf{wp}_\mathsf{b}\ C\ f$	$\mathsf{wlp}_\mathsf{b}\ C\ f$
skip	f	f
abort	$\mathbf{0}$	$\mathbf{1}$
$x := E$	$f[x/E]$	$f[x/E]$
observe (G)	$[G] \cdot f + [\neg G \wedge \mathsf{b}]$	$[G] \cdot f + [\neg G \wedge \mathsf{b}]$
$C_1; C_2$	$\mathsf{wp}_\mathsf{b}\ C_1\ (\mathsf{wp}_\mathsf{b}\ C_2\ f)$	$\mathsf{wlp}_\mathsf{b}\ C_1\ (\mathsf{wlp}_\mathsf{b}\ C_2\ f)$
ite $(G)\ \{C_1\}\ \{C_2\}$	$[G] \cdot \mathsf{wp}_\mathsf{b}\ C_1\ f + [\neg G] \cdot \mathsf{wp}_\mathsf{b}\ C_2\ f$	$[G] \cdot \mathsf{wlp}_\mathsf{b}\ C_1\ f + [\neg G] \cdot \mathsf{wlp}_\mathsf{b}\ C_2\ f$
$\{C_1\}\ [p]\ \{C_2\}$	$p \cdot \mathsf{wp}_\mathsf{b}\ C_1\ f + (1-p) \cdot \mathsf{wp}_\mathsf{b}\ C_2\ f$	$p \cdot \mathsf{wlp}_\mathsf{b}\ C_1\ f + (1-p) \cdot \mathsf{wlp}_\mathsf{b}\ C_2\ f$
while (G) $\{C'\}$	sup $F^n\ \mathbf{0}$, where	inf $F^n\ \mathbf{1}$, where
	$F\ X = [G] \cdot \mathsf{wp}_\mathsf{b}\ C'\ X + [\neg G] \cdot f$	$F\ X = [G] \cdot \mathsf{wlp}_\mathsf{b}\ C'\ X + [\neg G] \cdot f$

The parameter b controls whether or not to include the probability mass of observation failure. $\mathsf{wp}_{\mathbf{false}}$ coincides with the classic definition of wp (likewise for $\mathsf{wlp}_{\mathbf{false}}$), so we often omit the subscript when $b = \mathbf{false}$. The sup (inf) operation is defined with respect to the pointwise lifting to expectations of the standard order on $\mathbb{R}^\infty_{\geq 0}$ (i.e., $f \sqsubseteq g \iff \forall \sigma, f\ \sigma \leq g\ \sigma$ for expectations f and g).

Technically, only one of wp or wlp must be modified in this way, but in practice it is convenient to have both. The new definitions clearly subsume the old when $b = \mathbf{false}$, and importantly, satisfy the following symmetry conditions:

$$\mathsf{wp}_{\mathbf{false}}\ C\ \mathbf{0} = \mathbf{0}$$

$$\mathsf{wlp}_{\mathbf{true}}\ C\ \mathbf{1} = \mathbf{1}$$

$$\mathsf{wp}_\mathsf{b}\ C\ f + \mathsf{wlp}_{\neg\mathsf{b}}\ C\ (1-f) = \mathbf{1}, \text{ where } f \sqsubseteq \mathbf{1}.$$

The third of these conditions, the *invariant sum property*, has many useful implications. One immediate corollary is that for all C : cpGCL,

$$\mathsf{wlp}_{\mathbf{false}}\ C\ \mathbf{1} = \mathbf{1} - \mathsf{wp}_{\mathbf{true}}\ C\ \mathbf{0}.$$

and thus the usual definition of cwp (Definition 2) can be rewritten as follows:

$$\mathsf{cwp}\ C\ f \triangleq \frac{\mathsf{wp}_{\mathbf{false}}\ C\ f}{\mathbf{1} - \mathsf{wp}_{\mathbf{true}}\ C\ \mathbf{0}}$$

which corroborates a view of conditioning as simply wrapping the program with an outer "i.i.d." loop that resets the program to the beginning upon observation failure, as this expression of cwp has the form of the limit of a geometric series (see Section 2.5 for discussion of wp semantics of i.i.d. loops).

An analogue for cotrees of the above corollary is used in the proof of the CF tree equidistribution theorem from which the main equidistribution theorem (Theorem 10) for samplers compiled from cpGCL programs is derived.

3.4 cpGCL Formalized

We provide a Coq formalization of the cpGCL described in Section 2.3, with the following extensions:

1. We add to cpGCL an additional command for drawing samples uniformly at random from a range of natural numbers, and

2. we let probability expressions appearing in choice commands depend on the program state.

Definition 25 (cpGCL)**.** *The type* cpGCL *of commands in the conditional probabilistic guarded command language is defined inductively by the following formation rules:*

$$\frac{}{\textbf{skip} : \text{cpGCL}}\ \text{cpGCL-SKIP} \qquad \frac{x : \text{string} \qquad e : \Sigma \to \mathit{val}}{x \leftarrow e : \text{cpGCL}}\ \text{cpGCL-ASSIGN} \qquad \frac{c_1 : \text{cpGCL} \qquad c_2 : \text{cpGCL}}{c_1;\ c_2 : \text{cpGCL}}\ \text{cpGCL-SEQ}$$

$$\frac{e : \Sigma \to \mathbb{B}}{\textbf{observe}\ e : \text{cpGCL}}\ \text{cpGCL-OBSERVE} \qquad \frac{e : \Sigma \to \mathbb{B} \qquad c_1 : \text{cpGCL} \qquad c_2 : \text{cpGCL}}{\textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 : \text{cpGCL}}\ \text{cpGCL-ITE}$$

$$\frac{p : \Sigma \to \mathbb{Q} \qquad \forall \sigma : \Sigma,\ 0 \le p\,\sigma \le 1 \qquad c_1 : \text{cpGCL} \qquad c_2 : \text{cpGCL}}{\{\ c_1\ \}\ [p]\ \{\ c_2\ \} : \text{cpGCL}}\ \text{cpGCL-CHOICE}$$

$$\frac{e : \Sigma \to \mathbb{N} \qquad \forall \sigma : \Sigma,\ 0 < e\,\sigma \qquad k : \mathbb{N} \to \text{cpGCL}}{\textbf{uniform}\ e\ k : \text{cpGCL}}\ \text{cpGCL-UNIFORM} \qquad \frac{e : \Sigma \to \mathbb{B} \qquad c : \text{cpGCL}}{\textbf{while}\ e\ \textbf{do}\ c\ \textbf{end} : \text{cpGCL}}\ \text{cpGCL-WHILE}$$

The command '**uniform** $e\ k$' uniformly samples a natural number $0 \le n < e\,\sigma$ (where σ is the current program state) and continues execution with command '$k\ n$'.

Some PPLs provide a feature for "soft conditioning" via a command sometimes called **factor** [CTY06] or **score**. In contrast to "hard" conditioning (as supported in cpGCL by the **observe** command) which discards all executions not satisfying a given predicate, soft conditioning weights executions in the current branch by a given factor (so that weighting by 0 under an **if** command corresponds to hard conditioning). We derive a **score** command for soft conditioning from probabilistic choice and **observe**:

Definition 26 (**score** command)**.** *For expression* $e : \Sigma \to \mathbb{Q}$*, define* **score** e : cpGCL *as:*

$$\textbf{score}\ e \triangleq \{\ \textbf{skip}\ \}\ [e]\ \{\ \textbf{observe}\ \lambda\_.\ \textbf{false}\ \}.$$

3.5 cwp Formalized

We provide a Coq formalization of the generalized variants of wp and wlp described in Section 3.3, also extended with support for the **uniform** command.

Definition 27 (wp_b)**.** *For* $\mathsf{b} : \mathbb{B}$, $c : \mathsf{cpGCL}$, $f : \Sigma \to \mathbb{R}^\infty_{\geq 0}$, *and* $\sigma : \Sigma$, *define* $\mathsf{wp}_\mathsf{b}\ c\ f\ \sigma : \mathbb{R}^\infty_{\geq 0}$ *by induction on* c*:*

$\mathsf{wp}_\mathsf{b} : \mathsf{cpGCL} \to (\Sigma \to \mathbb{R}^\infty_{\geq 0}) \to \Sigma \to \mathbb{R}^\infty_{\geq 0}$

skip	$f \triangleq f$
$x \leftarrow e$	$f \triangleq f[x/e]$
observe e	$f \triangleq [e] \cdot f + [\neg e \wedge \mathsf{b}]$
$c_1;\ c_2$	$f \triangleq \mathsf{wp}_\mathsf{b}\ c_1\ (\mathsf{wp}_\mathsf{b}\ c_2\ f)$
if e **then** c_1 **else** c_2	$f \triangleq [e] \cdot \mathsf{wp}_\mathsf{b}\ c_1\ f + [\neg e] \cdot \mathsf{wp}_\mathsf{b}\ c_2\ f$
$\{\ c_1\ \}\ [p]\ \{\ c_2\ \}$	$f \triangleq p \cdot \mathsf{wp}_\mathsf{b}\ c_1\ f + (\mathbf{1} - p) \cdot \mathsf{wp}_\mathsf{b}\ c_2\ f$
uniform $e\ k$	$f \triangleq \lambda\sigma.\ \frac{1}{e\,\sigma} \sum_{i=0}^{e\,\sigma-1} \mathsf{wp}_\mathsf{b}\ (k\ i)\ f\ \sigma$
while e **do** c **end**	$f \triangleq \mathsf{sup}\ (F^n\ \mathbf{0})$, *where*
	$F\ g \triangleq [e] \cdot \mathsf{wp}_\mathsf{b}\ c\ g + [\neg e] \cdot f$

A collection of synonyms are used for different configurations of wp_b. The usual wp is recovered by setting $\mathsf{b} = \mathbf{false}$:

Definition 28 (wp)**.** *For command* $c : \mathsf{cpGCL}$, *expectation* $f : \Sigma \to \mathbb{R}^\infty_{\geq 0}$, *and program state* $\sigma : \Sigma$, *define* $\mathsf{wp}\ c\ f\ \sigma : \mathbb{R}^\infty_{\geq 0}$ *(the expected value of* f *when running* c *from initial state* σ*) by:*

$$\mathsf{wp}\ c\ f\ \sigma : \mathbb{R}^\infty_{\geq 0} \triangleq \mathsf{wp}_{\mathbf{false}}\ c\ f\ \sigma.$$

wpfail bundles the probability mass of observation failure together with the expected value of f by setting $b = \mathbf{true}$:

Definition 29 (wpfail)**.** *For command* $c : \mathsf{cpGCL}$*, expectation* $f : \Sigma \to \mathbb{R}^{\infty}_{\geq 0}$*, and program state* $\sigma : \Sigma$*, define* $\mathsf{wpfail}\ c\ f\ \sigma : \mathbb{R}^{\infty}_{\geq 0}$ *(the expected value of* f *when running* c *from initial state* σ *plus the probability mass of observation failure) by:*

$$\mathsf{wpfail}\ c\ f\ \sigma : \mathbb{R}^{\infty}_{\geq 0} \triangleq \mathsf{wp}_{\mathbf{true}}\ c\ f\ \sigma.$$

The probability of observation failure is then given by wpfail when f is constant at 0:

Definition 30 (fail)**.** *For command* $c : \mathsf{cpGCL}$ *and program state* $\sigma : \Sigma$*, define* $\mathsf{fail}\ c\ \sigma : \mathbb{R}^{\leq 1}_{\geq 0}$ *(the probability of observation failure) by:*

$$\mathsf{fail}\ c\ \sigma : \mathbb{R}^{\leq 1}_{\geq 0} \triangleq \mathsf{wpfail}\ c\ \mathbf{0}\ \sigma.$$

Definition 31 ($\mathsf{wlp_b}$)**.** *For* $b : \mathbb{B}$*,* $c : \mathsf{cpGCL}$*,* $f : \Sigma \to \mathbb{R}^{\leq 1}_{\geq 0}$ *(a bounded expectation), and* $\sigma : \Sigma$*, define* $\mathsf{wlp_b}\ c\ f\ \sigma : \mathbb{R}^{\leq 1}_{\geq 0}$ *by induction on* c *(we list only the* **while** *case as the rest are like* wp*, mutatis mutandis):*

$$\mathsf{wlp_b} : \mathsf{cpGCL} \to (\Sigma \to \mathbb{R}^{\leq 1}_{\geq 0}) \to \Sigma \to \mathbb{R}^{\leq 1}_{\geq 0}$$

$$\textbf{while}\ e\ \textbf{do}\ c\ \textbf{end}\ \ f \triangleq \mathsf{inf}\ (F^n\ \mathbf{1}),\ \textit{where}$$

$$F\ g \triangleq [e] \cdot \mathsf{wlp_b}\ c\ g + [\neg e] \cdot f$$

The usual wlp is recovered from $\mathsf{wlp_b}$ by setting b = **false**:

Definition 32 (wlp)**.** *For command* $c : \mathsf{cpGCL}$*, bounded expectation* $f : \Sigma \to \mathbb{R}^{\leq 1}_{\geq 0}$*, and program state* $\sigma : \Sigma$*, define* $\mathsf{wlp}\ c\ f\ \sigma : \mathbb{R}^{\leq 1}_{\geq 0}$ *(the expected value of* f *when running* c *from initial state* σ *plus the probability mass of divergence) by:*

$$\mathsf{wlp}\ c\ f\ \sigma : \mathbb{R}^{\leq 1}_{\geq 0} \triangleq \mathsf{wlp}_{\mathbf{false}}\ c\ f\ \sigma.$$

wlpfail bundles the probability mass of observation failure together with the liberal expected value of f by setting b = **true**:

Definition 33 (wlpfail)**.** *For command* $c : \mathsf{cpGCL}$*, bounded expectation* $f : \Sigma \to \mathbb{R}^{\leq 1}_{\geq 0}$*, and program state* $\sigma : \Sigma$*, define* $\mathsf{wlpfail}\ c\ f\ \sigma : \mathbb{R}^{\leq 1}_{\geq 0}$ *(the expected value of* f *when running* c *from initial state* σ *plus the probability masses of observation failure and divergence) by:*

$$\mathsf{wlpfail}\ c\ f\ \sigma : \mathbb{R}^{\leq 1}_{\geq 0} \triangleq \mathsf{wlp}_{\mathbf{true}}\ c\ f\ \sigma.$$

The probability of failure *or* divergence is obtained from wlpfail by specializing f to $\mathbf{0}$:

Definition 34 (fail_or_diverge)**.** *For command* $c : \mathsf{cpGCL}$ *and program state* $\sigma : \Sigma$*, define* $\mathsf{fail\_or\_diverge}\ c\ \sigma : \mathbb{R}^{\leq 1}_{\geq 0}$ *(the probability of observation failure plus the probability of divervgence) by:*

$$\mathsf{fail\_or\_diverge}\ c\ \sigma : \mathbb{R}^{\leq 1}_{\geq 0} \triangleq \mathsf{wlpfail}\ c\ \mathbf{0}\ \sigma.$$

Lastly, cwp is defined in the obvious way to match Definition 2:

Definition 35 (cwp)**.** *For command* $c : \mathsf{cpGCL}$ *and expectation* $f : \Sigma \to \mathbb{R}^{\infty}_{\geq 0}$*,*

$$\mathsf{cwp}\ c\ f : \Sigma \to \mathbb{R}^{\infty}_{\geq 0} \triangleq \frac{\mathsf{wp}_{\mathbf{false}}\ c\ f}{\mathsf{wlp}_{\mathbf{false}}\ c\ \boldsymbol{1}}.$$

The next two lemmas prove that wp is continuous and that wlp is l-continuous:

Lemma 1 (wp is continuous)**.** *Let* $b : \mathbb{B}$*,* $c : \mathsf{cpGCL}$*, and let* $f : \mathbb{N} \to \Sigma \to \mathbb{R}^{\infty}_{\geq 0}$ *be an* ω*-chain of expectations. Then,*

$$\mathsf{wp}_{\mathsf{b}}\ c\ (\mathsf{sup}\ f) = \mathsf{sup}\ (\mathsf{wp}_{\mathsf{b}}\ c \circ f).$$

Lemma 2 (wlp is l-continuous)**.** *Let* $b : \mathbb{B}$*,* $c : \mathsf{cpGCL}$*, and let* $f : \mathbb{N} \to \Sigma \to \mathbb{R}^{\infty}_{\geq 0}$ *be a decreasing* ω*-chain of expectations. Then,*

$$\mathsf{wlp}_{\mathsf{b}}\ c\ (\mathsf{inf}\ f) = \mathsf{inf}\ (\mathsf{wlp}_{\mathsf{b}}\ c \circ f).$$

We also validate our definition of cwp by checking that loops are semantically equivalent to their one-step unrollings (as we noted to expect in Section 2.5):

Theorem 1 (cwp of loops)**.** *Let* $G : \Sigma \to \mathbb{B}$*,* $c :$ cpGCL*, and* $f : \Sigma \to \mathbb{R}^{\infty}_{\geq 0}$*. Then,*

$$\mathsf{cwp}\ (\textbf{while}\ G\ \textbf{do}\ c\ \textbf{end})\ f = \mathsf{cwp}\ (\textbf{if}\ G\ \textbf{then}\ c;\ \textbf{while}\ G\ \textbf{do}\ c\ \textbf{end}\ \textbf{else skip})\ f.$$

4 Compiling cpGCL

We are all hungry and thirsty for
concrete images.

Salvador Dalí

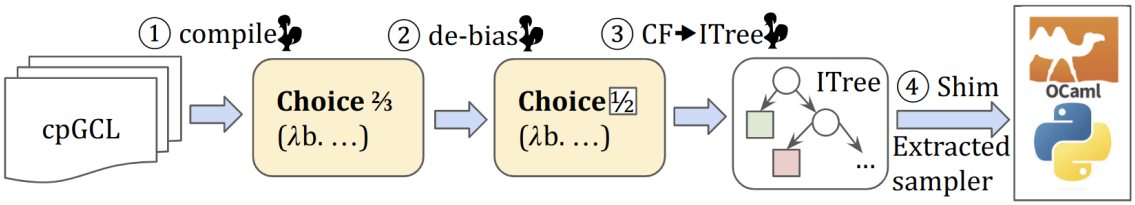


Figure 4.1: Zar compiler pipeline.

In this chapter we describe the strategy employed by Zar for compiling high-level cpGCL programs to interaction tree samplers for efficient execution in OCaml. We introduce *choice-fix* (CF) trees as an intermediate program representation (Section 4.1) and define a weakest pre-expectation style semantics over them (Section 4.2). We then show how cpGCL programs are compiled to CF trees (Section 4.3) and describe an algorithm for transforming CF trees containing biased probabilistic choices to semantically equivalent CF trees with only unbiased (with probability $\frac{1}{2}$ of taking either branch) choices (Section 4.4). Finally, we show how the sampling processes encoded by unbiased CF trees are elaborated to an executable interaction tree representation (Section 4.5).

4.1 Choice-Fix (CF) Trees

Toward the ultimate goal of compiling cpGCL programs to proved-correct executable samplers, we introduce a tree-based intermediate representation called *choice-fix* (CF) trees. Inspired by the discrete distribution generating (DDG) trees introduced by Knuth and Yao [KY76], CF trees can be understood as encoding the operational sampling

behavior of probabilistic programs as decision processes with probabilistic branching and loops. Probabilistic choices appearing in DDG trees typically correspond to fair coin flips, i.e., the Bernoulli distribution with $p = \frac{1}{2}$. CF trees differ in this regard; probabilistic choices may be *biased* with bias parameter $p \in [0, 1]$ denoting the probability of "heads" or "taking the left subtree".

CF trees occupy a sweet-spot of abstraction between cpGCL programs and the coinductive backends of interactions trees (Section 4.5) and algebraic cotrees (Chapter 7), serving as a convenient compilation target from cpGCL as well as an inductive notation for potentially infinite binary trees that can be naturally "unfolded" to a coinductive representation for execution. CF trees are also a convenient representation for program transformations such as de-biasing (Section 4.4), which are necessary for generating samplers that operate within the random bit model.

Definition 36 (CF trees)**.** *Let $\mathcal{T}^{cf}_{\Sigma}$ (the type of CF trees with element type Σ) be the smallest collection of elements closed under the following formation rules:*

$$\frac{\sigma : \Sigma}{\textbf{leaf}\ \sigma : \mathcal{T}^{cf}_{\Sigma}}\ \text{CF-LEAF} \qquad \frac{}{\textbf{fail} : \mathcal{T}^{cf}_{\Sigma}}\ \text{CF-FAIL} \qquad \frac{p : \mathbb{Q} \qquad 0 \le p \le 1 \qquad k : \mathbb{B} \to \mathcal{T}^{cf}_{\Sigma}}{\textbf{choice}\ p\ k : \mathcal{T}^{cf}_{\Sigma}}\ \text{CF-CHOICE}$$

$$\frac{\sigma : \Sigma \qquad e : \Sigma \to \mathbb{B} \qquad g : \Sigma \to \mathcal{T}^{cf}_{\Sigma} \qquad k : \Sigma \to \mathcal{T}^{cf}_{\Sigma}}{\textbf{fix}\ \sigma\ e\ g\ k : \mathcal{T}^{cf}_{\Sigma}}\ \text{CF-FIX}$$

The name "choice-fix" is due to the two non-leaf constructors of the type: 1) *choice* nodes for probabilistic choice, and 2) *fix* nodes for encoding loops. '**leaf** σ' denotes the end of a program execution with terminal state $\sigma : \Sigma$. '**fail**' denotes a program execution in which an observed predicate (via the **observe** command) has been violated. '**choice** $p\ k$' represents a probabilistic binary choice between two subtrees 'k **true**' and

‘k **false**’ where rational bias $p \in [0, 1]$ denotes the probability of taking the left subtree ‘k **true**’) and $1 - p$ the probability of taking the right subtree ‘k **false**’.

Lastly, ‘**fix** σ e g k’ encodes a loop with initial state σ, guard condition e, body generator g, and continuation k, and should be understood operationally as follows: Starting with initial CF tree ‘**leaf** σ’, repeatedly extend the leaves of the tree constructed thus far via either the generating function g (re-entering the loop when $e\ \sigma$ = **true**) or continuation k (exiting the loop when $e\ \sigma$ = **false**). Figure 4.2 shows the CF tree representation of the ‘primes’ program from Figure 1.1a.

```
choice p (λb₀.
  if b₀ then
    fix {h ↦ 0, b ↦ true} (λσ. σ[b])
      (λσ. choice p (λb′. if b′
        then leaf (σ[h ↦ h + 1, b ↦ true])
        else leaf (σ[h ↦ h + 1, b ↦ false])))
      (λσ. if σ h is prime then leaf σ else fail)
  else fail)
```

Figure 4.2: CF tree term representation of Program 1.1a.

The type $\mathcal{T}^{cf}_{\Sigma}$ forms a *monad* [Wad92] (a variant of the tree monad [JD93]), with monadic return given by the **leaf** constructor, and bind (essential for compiling sequenced cpGCL commands) given by Def. 37 (using the infix notation ’$\gg\!=_{\mathsf{cf}}$’).

Definition 37 (CF tree bind)**.** *For CF tree $t : \mathcal{T}^{cf}_{\Sigma}$ and $\mathcal{T}^{cf}_{\Sigma}$-valued continuation $k : \Sigma \to \mathcal{T}^{cf}_{\Sigma}$, define $t \gg\!= k : \mathcal{T}^{cf}_{\Sigma}$ by induction on t:*

$\gg\!=_{\mathsf{cf}} : \mathcal{T}^{cf}_{\Sigma} \to (\Sigma \to \mathcal{T}^{cf}_{\Sigma}) \to \mathcal{T}^{cf}_{\Sigma}$

leaf σ	$\gg\!=_{\mathsf{cf}}$	k	$\triangleq\ k\ \sigma$
fail	$\gg\!=_{\mathsf{cf}}$	_	$\triangleq$ **fail**
choice $p\ f$	$\gg\!=_{\mathsf{cf}}$	k	$\triangleq$ **choice** $p\ (\lambda b.\ f\ b \gg\!=_{\mathsf{cf}} k)$
fix $e\ g\ h$	$\gg\!=_{\mathsf{cf}}$	k	$\triangleq$ **fix** $e\ g\ (\lambda\sigma.\ h\ \sigma \gg\!=_{\mathsf{cf}} k)$

Definition 37 is useful for compiling sequential compositions of cpGCL commands to CF trees in Section 4.3.

4.2 CF Tree Semantics

The *inference* (or *twp*) semantics of CF trees is defined analogously to the cwp semantics of cpGCL (Section 3.5). The expression '$\mathsf{twp}_{\mathbf{false}}\ t\ f$' denotes the expected value of expectation f over CF tree t. When $\mathsf{b} = \mathbf{true}$, $\mathsf{twp}_{\mathsf{b}}$ additionally includes the probability mass of observation failure.

Definition 38 ($\mathsf{twp}_{\mathsf{b}}$)**.** *For $t : \mathcal{T}^{cf}_{\Sigma}$ a CF tree and $f : \Sigma \to \mathbb{R}^{\infty}_{\geq 0}$ an expectation, define $\mathsf{twp}_{\mathsf{b}}\ t\ f : \mathbb{R}^{\infty}_{\geq 0}$ by induction on t:*

$$\mathsf{twp}_{\mathsf{b}} : \mathcal{T}^{cf}_{\Sigma} \to (\Sigma \to \mathbb{R}^{\infty}_{\geq 0}) \to \mathbb{R}^{\infty}_{\geq 0}$$

$$\begin{aligned}
&\mathbf{leaf}\ \sigma && f \triangleq f\ \sigma \\
&\mathbf{fail} && _ \triangleq [\mathsf{b}] \\
&\mathbf{choice}\ p\ k && f \triangleq p \cdot \mathsf{twp}_{\mathsf{b}}\ (k\ \mathbf{true})\ f + (1-p) \cdot \mathsf{twp}_{\mathsf{b}}\ (k\ \mathbf{false})\ f \\
&\mathbf{fix}\ \sigma_0\ e\ g\ k && f \triangleq \mathsf{sup}\ (F^n\ \mathbf{0})\ \sigma_0, \textit{ where} \\
& && \quad F\ h\ \sigma \triangleq \textit{if } e\ \sigma \textit{ then } \mathsf{twp}_{\mathsf{b}}\ (g\ \sigma)\ h \textit{ else } \mathsf{twp}_{\mathsf{b}}\ (k\ \sigma)\ f
\end{aligned}$$

The "liberal" variant of twp is defined as follows, where '$\mathsf{twlp}_{\mathsf{b}}\ t\ f$' denotes the expected value of expectation f over CF tree t plus the probability mass of divergence (and plus the mass of observation failure when $b = \mathbf{true}$).

Definition 39 (twlp$_b$). *For $t : \mathcal{T}^{cf}_{\Sigma}$ a CF tree and $f : \Sigma \to \mathbb{R}^{\leq 1}_{\geq 0}$ a bounded expectation, define* $\mathsf{twlp}_b\ t\ f : \mathbb{R}^{\leq 1}_{\geq 0}$ *by induction on t:*

$\mathsf{twlp}_b : \mathcal{T}^{cf}_{\Sigma} \to (\Sigma \to \mathbb{R}^{\leq 1}_{\geq 0}) \to \mathbb{R}^{\leq 1}_{\geq 0}$			
leaf σ	f	$\triangleq$	$f\ \sigma$
fail	$\_$	$\triangleq$	$[\mathsf{b}]$
choice $p\ k$	f	$\triangleq$	$p \cdot \mathsf{twlp}_b\ (k\ \mathbf{true})\ f + (1-p) \cdot \mathsf{twlp}_b\ (k\ \mathbf{false})\ f$
fix $\sigma_0\ e\ g\ k$	f	$\triangleq$	$\mathsf{inf}\ (F^n\ \mathbf{1})\ \sigma_0$, *where*
			$F\ h\ \sigma \triangleq$ *if* $e\ \sigma$ *then* $\mathsf{twlp}_b\ (g\ \sigma)\ h$ *else* $\mathsf{twlp}_b\ (k\ \sigma)\ f$

The conditional (tcwp) semantics for CF trees then matches cwp (Definition 2):

Definition 40 (tcwp). *For $t : \mathcal{T}^{cf}_{\Sigma}$ a CF tree and $f : \Sigma \to \mathbb{R}^{\infty}_{\geq 0}$ an expectation, define* $\mathsf{tcwp}\ t\ f : \mathbb{R}^{\infty}_{\geq 0}$ *by:*

$$\mathsf{tcwp}\ t\ f : \mathbb{R}^{\infty}_{\geq 0} \triangleq \frac{\mathsf{twp}_{\mathbf{false}}\ t\ f}{\mathsf{twlp}_{\mathbf{false}}\ t\ \mathbf{1}}.$$

Many desirable properties of wp and wlp also hold in the CF tree setting. Notably, we have an analogue of the invariant sum property:

Theorem 2 (CF tree invariant sum). *Let $t : \mathcal{T}^{cf}_{\Sigma}$ be a well-formed CF tree, $b : \mathbb{B}$, and $f : \Sigma \to \mathbb{R}^{\leq 1}_{\geq 0}$ a bounded expectation. Then,*

$$\mathsf{twp}_b\ t\ f + \mathsf{twlp}_{\neg b}\ t\ (\mathbf{1} - f) = 1.$$

Our intent is that the tcwp semantics of the CF tree representation of a cpGCL program should coincide exactly with the program's cwp semantics. Indeed, the following section presents a semantics-preserving compiler from cpGCL to CF trees.

4.3 Compiling to CF Trees

A command $c : \mathsf{cpGCL}$ is compiled to a function $[\![c]\!] : \Sigma \to \mathcal{T}^{cf}_{\Sigma}$ mapping initial state $\sigma : \Sigma$ to the CF tree encoding the sampling semantics of c starting from σ. Recall that the

operator '$\gg\!=$' denotes bind in the CF tree monad (Definition 37). The algorithm in Definition 41 performs the translation from cpGCL programs to their corresponding CF tree representations.

Definition 41 ($[\![\cdot]\!]$). *For command* $c : \mathsf{cpGCL}$ *and program state* $\sigma : \Sigma$, *define* $[\![c]\!]\,\sigma : \mathcal{T}^{cf}_{\Sigma}$ *by induction on* c*:*

$[\![\cdot]\!] : \mathsf{cpGCL} \to \Sigma \to \mathcal{T}^{cf}_{\Sigma}$		
skip	σ	$\triangleq$ **leaf** σ
$x \leftarrow e$	σ	$\triangleq$ **leaf** $\sigma[x \mapsto e\,\sigma]$
observe e	σ	$\triangleq$ *if* $e\,\sigma$ *then* **leaf** σ *else* **fail**
$c_1;\ c_2$	σ	$\triangleq$ $[\![c_1]\!]\,\sigma \gg\!= [\![c_2]\!]$
if e **then** c_1 **else** c_2	σ	$\triangleq$ *if* $e\,\sigma$ *then* $[\![c_1]\!]\,\sigma$ *else* $[\![c_2]\!]\,\sigma$
$\{\ c_1\ \}\ [e]\ \{\ c_2\ \}$	σ	$\triangleq$ **choice** $(e\,\sigma)$ $(\lambda b.$ *if* b *then* $[\![c_1]\!]\,\sigma$ *else* $[\![c_2]\!]\,\sigma)$
uniform $e\ k$	σ	$\triangleq$ $\mathsf{uniform\_tree}\ (e\,\sigma) \gg\!= \lambda n.\ [\![k\ n]\!]\,\sigma$
while e **do** c **end**	σ	$\triangleq$ **fix** $\sigma\ e\ [\![c]\!]$ **leaf**

The function '$\mathsf{uniform\_tree}$' builds a CF tree encoding a uniform distribution over a fixed range of natural numbers, the specification of which is captured by the following lemma:

Lemma 3 (Uniform tree correctness). *Let* $0 < n : \mathbb{N}$ *and* $f : \mathbb{N} \to \mathbb{R}^{\infty}_{\geq 0}$. *Then,*

$$\mathsf{twp}_{\mathbf{false}}\ (\mathsf{uniform\_tree}\ n)\ f = \frac{1}{n}\sum_{i=0}^{n-1} f\ i.$$

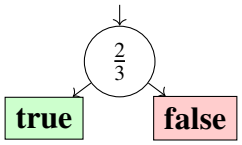
By setting $f = [\lambda m.\ m = k]$, we obtain as an immediate consequence of Lemma 3 that $\mathsf{twp}_{\mathbf{false}}\ (\mathsf{uniform\_tree}\ n)\ [\lambda m.\ m = k] = \frac{1}{n}$ for all $k < n$, or in other words, that $\mathsf{uniform\_tree}$ is uniformly distributed. The overall compiler is then proved correct by the following theorem establishing the correspondence of the cwp semantics of cpGCL programs with the tcwp semantics of the CF trees generated from them.

Theorem 3 (CF tree compiler correctness)**.** *Let* $c : \mathsf{cpGCL}$*,* $f : \Sigma \to \mathbb{R}^{\infty}_{\geq 0}$*, and* $\sigma : \Sigma$*. Then,*

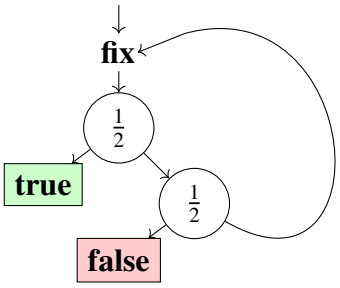
$$\mathsf{tcwp}\ (\llbracket c \rrbracket\ \sigma)\ f = \mathsf{cwp}\ c\ f\ \sigma.$$

4.4 De-Biasing CF Trees

CF trees generated by the Zar compiler may have arbitrary $p \in [0, 1] \subseteq \mathbb{Q}$ bias values at choice nodes. To obtain sampling procedures in the random bit model, we apply a bias-elimination transformation to replace all probabilistic choices in the generated CF trees by fair coin-flipping schemes that implement semantically equivalent behavior. The CF trees resulting from this transformation have $p = \frac{1}{2}$ at all choice nodes. Figure 4.3 shows an example of the debiasing transformation applied to a **choice** node with $p = \frac{2}{3}$.



(a) Biased choice with Pr(**true**) = $\frac{2}{3}$



(b) Debiased tree with Pr(**true**) = $\frac{2}{3}$

Figure 4.3: **choice** CF tree with $Pr(\textbf{true}) = \frac{2}{3}$ (left) and corresponding debiased CF tree (right).

The algorithm for translating a '**choice** $p\ k$' node with rational bias $p = \dfrac{n}{d}$ and subtrees $t_1 = k\ \textbf{true}$ and $t_2 = k\ \textbf{false}$ goes as follows:

1. Recursively translate t_1 and t_2, yielding t_1' and t_2' respectively,
2. choose $m : \mathbb{N}$ such that $2^{m-1} < d \leq 2^m$,

3. generate a perfect CF tree of depth m with all terminal nodes marked by a special **loopback** value,

4. replace the first n terminals with copies of subtree t'_1, and the next d terminals with copies of subtree t'_2, leaving $s^m - n - d$ **loopback** nodes remaining,

5. coalesce duplicate leaf nodes to eliminate redundancy,

6. wrap the tree in a **fix** constructor with guard condition that evaluates to true on the **loopback** value, and

7. replace the original **choice** node with the newly generated tree.

In essence, the biased choice is replaced by a rejection sampler that simulates a biased coin by repeated flips of a fair one. An implementation of the choice translation algorithm is in the file 'uniform.v' under the name 'bernoulli_tree'. As its name suggests, bernoulli_tree takes a rational probability p and produces a CF tree over Boolean outputs with probability p of **true**. The overall debiasing transformation on CF trees is then a straightforward recursive traversal of the input tree, using bernoulli_tree in conjunction with monadic bind (Definition 37) to replace biased **choice** nodes with equivalent subtrees containing only unbiased choices.

Definition 42 (debias)**.** *For $t : \mathcal{T}_\Sigma^{cf}$ a CF tree, define* debias $t : \mathcal{T}_\Sigma^{cf}$ *by induction on t:*

debias : $\mathcal{T}_\Sigma^{cf} \to \mathcal{T}_\Sigma^{cf}$

leaf σ $\triangleq$ **leaf** σ

fail $\triangleq$ **fail**

choice $p\ k$ $\triangleq$ bernoulli_tree p $\gg\!=$ λb. *if* b *then* debias (k **true**) *else* debias (k **false**)

fix $\sigma\ e\ g\ k$ $\triangleq$ **fix** $\sigma\ e$ (debias $\circ\ g$) (debias $\circ\ k$)

The essential results for the debiasing transformation now follow: debias preserves tcwp semantics and produces trees in which all choices are unbiased.

Theorem 4 (debias preserves tcwp semantics)**.** *Let* $t : \mathcal{T}_\Sigma^{cf}$ *be a CF tree and* $f : \Sigma \to \mathbb{R}_{\geq 0}^{\infty}$ *an expectation. Then,*

$$\mathsf{tcwp}\ (\mathsf{debias}\ t)\ f = \mathsf{tcwp}\ t\ f.$$

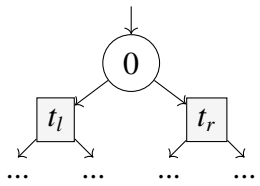
Theorem 5 (debias produces unbiased trees)**.** *Let* $t : \mathcal{T}_\Sigma^{cf}$ *be a CF tree. Then,* $p = \frac{1}{2}$ *for every '* **choice** $p\ k$*' in* debias t*.*

CF Tree Optimization A potential cause of inefficiency of the de-biasing transformation is the existence of redundant choice nodes. For example, when $p = 0$, a '**choice** $p\ k$' node can never possibly take its left branch and so may as well be outright replaced by its right subtree 'k **false**' instead of being wrapped in an unnecessary bernoulli_tree construction (illustrated in Figure 4.4a). Likewise, when $p = 1$, '**choice** $p\ k$' ought to be replaced by its left subtree 'p **true**' (Figure 4.4b). Additionally, the type Q of rational numbers in Coq is not canonically represented (i.e., not guaranteed to be in reduced form), and non-reduced rationals may incur unnecessary overhead in the construction of bernoulli_trees. To resolve both of these issues, we implement an optimization pass (applied *before* debiasing) that eliminates redundant choices and reduces rational probabilities via the function Qred : $\mathbb{Q} \to \mathbb{Q}$ from the Coq standard library.

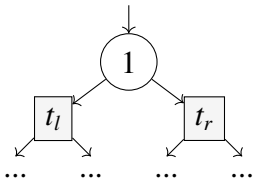
Definition 43 (elim_choices)**.** *For* $t : \mathcal{T}_\Sigma^{cf}$ *a CF tree, define* elim_choices $t : \mathcal{T}_\Sigma^{cf}$ *by induction on* t*:*

elim_choices : $\mathcal{T}_\Sigma^{cf} \to \mathcal{T}_\Sigma^{cf}$

$$\begin{aligned}
&\mathbf{leaf}\ \sigma && \triangleq \mathbf{leaf}\ \sigma \\
&\mathbf{fail} && \triangleq \mathbf{fail} \\
&\mathbf{choice}\ p\ k && \triangleq \mathit{if}\ p = 0\ \mathit{then}\ \mathsf{elim_choices}\ (k\ \mathbf{false})\ \mathit{else} \\
& && \quad \mathit{if}\ p = 1\ \mathit{then}\ \mathsf{elim_choices}\ (k\ \mathbf{true})\ \mathit{else} \\
& && \quad \mathbf{choice}\ (\mathsf{Qred}\ p)\ (\mathsf{elim_choices} \circ k) \\
&\mathbf{fix}\ \sigma\ e\ g\ k && \triangleq \mathbf{fix}\ \sigma\ e\ (\mathsf{elim_choices} \circ g)\ (\mathsf{elim_choices} \circ k)
\end{aligned}$$



(a) **choice** p k node with $p = 0$, k **true** $= t_l$, and k **false** $= t_r$. Since the probability of taking t_l is 0, the entire node can be replaced by the right subtree t_r.



(b) **choice** p k node with $p = 1$, k **true** $= t_l$, and k **false** $= t_r$. The entire node can be replaced by the left subtree t_l.

Figure 4.4: Illustration of redundant **choice** nodes.

We prove that elim_choices preserves tcwp semantics, and produces CF trees such that for all $p : \mathbb{Q}$ appearing in choice nodes, $0 < p < 1$ and p is in reduced form.

Theorem 6 (elim_choices preserves tcwp semantics)**.** *Let $t : \mathcal{T}_\Sigma^{cf}$ be a CF tree and $f : \Sigma \to \mathbb{R}^\infty_{\geq 0}$ an expectation. Then,*

$$\mathsf{tcwp}\ (\mathsf{elim\_choices}\ t)\ f = \mathsf{tcwp}\ t\ f.$$

Theorem 7 (elim_choices produces reduced trees)**.** *Let $t : \mathcal{T}_\Sigma^{cf}$ be a CF tree and $f : \Sigma \to \mathbb{R}^\infty_{\geq 0}$ an expectation. Then, p is in reduced form and $0 < p < 1$ for every '**choice** p k' node in t.*

De-biased CF trees are close to being executable samplers in the random bit model. However, since they permit the existence of infinite execution paths (and hence denote sampling processes that can't be expected in general to always terminate), we must first pass to the coinductive unfolding of CF trees for execution.

We provide two target coinductive representations. One of them, based on the concept of *algebraic CPO* and built on the AlgCo framework (the topic of Chapter 6), is

developed in Chapter 7. It is this representation on which we can naturally perform weakest pre-expectation style reasoning.

The other representation, primarily for purposes of extraction and execution, is based on the interaction tree library [XZH+20] and makes use of the paco library [HNDV13] for coinductive reasoning. We present this interaction tree representation and the algorithm for generating interaction trees from CF trees in the following section (Section 4.5).

4.5 Generating Interaction Trees

Interaction trees [XZH+20] (ITrees) are a general-purpose coinductive data structure for modeling effectful (co-)recursive programs that interact with their environments. The coq-itree library provides a suite of combinators for constructing ITrees, along with a collection of formal principles for reasoning about their equivalence. An interaction tree computation performs an effect by raising an event (which may carry data) that is then handled by the environment, possibly providing data in return. In this section we show how to generate executable interaction tree samplers from CF trees.

Interaction Tree Syntax Interaction trees are parameterized by an event functor $E : \mathsf{Type} \to \mathsf{Type}$ that specifies the kinds of interactions the computation can have with its environment. In our case, there is only one kind of interaction: a sampler may query its environment for a single randomly generated bit. Thus the event functor $\mathsf{boolE} : \mathsf{Type} \to \mathsf{Type}$ has just one constructor **get** taking zero arguments, with type index $\mathbb{B}$ indicating that the environment's response should be a single Boolean value.

Definition 44 ($\mathcal{T}_A^{it}$)**.** *Define the type $\mathcal{T}_A^{it}$ of interaction trees with event functor* boolE *and element type A coinductively by the formation rules:*

$$\frac{}{\textbf{get} : \mathsf{boolE}\ \mathbb{B}}\ \textsc{boolE-get} \qquad \frac{a : A}{\textbf{ret}\ a : \mathcal{T}_A^{it}}\ \textsc{itree-ret} \qquad \frac{t : \mathcal{T}_A^{it}}{\textbf{tau}\ t : \mathcal{T}_A^{it}}\ \textsc{itree-tau} \qquad \frac{k : \mathbb{B} \to \mathcal{T}_A^{it}}{\textbf{vis get}\ k : \mathcal{T}_A^{it}}\ \textsc{itree-vis}$$

The constructors of $\mathcal{T}_A^{it}$ encode a sampling procedure over sample space A as follows: '**ret** a' produces the sample $a : A$, '**tau** t' performs a single "silent step" (see 6.1 for discussion) producing no effect and continuing the sampling process with $t : \mathcal{T}_A^{it}$, and '**vis get** k' requests a single bit $b : \mathbb{B}$ from the environment and continues with '$k\ b : \mathcal{T}_A^{it}$'. For example, Figure 4.5 shows a coinductively defined ITree encoding a sampler for the $\mathsf{Bernoulli}(\frac{2}{3})$ distribution corresponding to the CF tree in Figure 4.3b.

$$
\begin{aligned}
it_{\frac{2}{3}} \triangleq\ & \textbf{vis get}\ (\lambda b_0.\ \text{if}\ b_0\ \text{then}\ \textbf{ret true} \\
& \qquad \text{else}\ \textbf{vis get}\ (\lambda b_1.\ \text{if}\ b_1\ \text{then}\ \textbf{ret false} \\
& \qquad\qquad \text{else}\ \textbf{tau}\ it_{\frac{2}{3}}))
\end{aligned}
$$

Figure 4.5: ITree encoding of $\mathsf{Bernoulli}(\frac{2}{3})$ distribution corresponding to the CF tree in Figure 4.3b. The corecursive occurrence of $it_{\frac{2}{3}}$ is guarded by a **tau** constructor, a common practice (although not necessary in this example) to ensure productivity of the ITree (see Section 6.1 for related discussion on **tau** nodes).

Unfolding a CF tree to an interaction tree proceeds in two steps:

1. Generate an "open" ITree $t : \mathcal{T}_{(\mathbf{1}+\Sigma)}^{it}$ by induction on the input CF tree (with **1** on the LHS encoding observation failure), and then
2. "tie the knot" through observation failures in t to produce the final ITree of type $\mathcal{T}_{\Sigma}^{it}$.

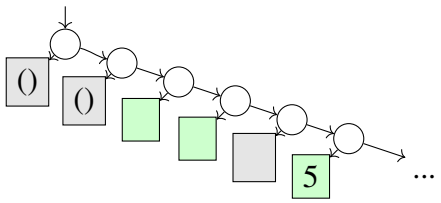
The first step is implemented by the function $\mathsf{to\_itree\_open}$ (see Figure 4.6a):

Definition 45 (to_itree_open)**.** *Given an unbiased CF tree* $t : \mathcal{T}^{cf}_{\Sigma}$*, define* to_itree_open $t : \mathcal{T}^{it}_{\mathbf{1}+\Sigma}$ *by induction on t:*

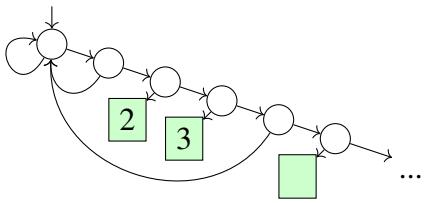
```
to_itree_open : T^cf_Σ → T^it_{1+Σ}
leaf σ      ≜ ret (inr σ)
fail        ≜ ret (inl ())
choice _ k  ≜ vis get (to_itree_open ∘ k)
fix σ0 e g k ≜ ITree.iter (λσ. if e σ then y ← to_itree_open (g σ) ;;
                                  match y with
                                  | inl () ⇒ ret (inr (inl ()))
                                  | inr σ' ⇒ ret (inl σ')
                                  end
                               else ITree.map inr (to_itree_open (k σ))) σ0
```

Since ITrees have just one kind of terminal constructor (**ret**) to CF trees' two (**leaf** and **fail**), we generate ITrees of type $\mathcal{T}^{it}_{(\mathbf{1}+\Sigma)}$ where **fail** nodes are translated to **ret** (inl ()), and nodes of the form **leaf** x to **ret** (inr x), where inl and inr are the left and right sum injections. **fix** nodes are translated via application of the ITree.iter combinator. We refer to [XZH+20, Section 4] for a general explanation of iteration with ITrees; for us it is enough to know that ITree.iter takes a generating function of type $\Sigma \to \mathcal{T}^{it}_{(\Sigma+(\mathbf{1}+\Sigma))}$ and "ties the knot" through Σ on the LHS to produce an ITree of type $\mathcal{T}^{it}_{(\mathbf{1}+\Sigma)}$. The generating function encodes the body of the loop as well as its continuation, where returning inl σ indicates to repeat the loop with a new iteration generated from state σ, and returning inr x with $x : \mathbf{1} + \Sigma$ indicates to terminate the program with value x which is either inl () for observation failure or inr σ for terminal state σ (see Figure 4.6a).

The second step is implemented by the following definition of 'tie_itree' which corecursively "ties the knot" [Elk21] through the left side of the sum $\mathbf{1} + \Sigma$ via ITree.iter to



(a) to_itree_open primes



(b) tie_itree (to_itree_open primes)

Figure 4.6: Interaction trees generated by 'to_itree_open primes' (left) and then by "tying the knot" via 'tie_itree' (right), where 'primes' is the cpGCL program in Figure 1.1a.

produce ITree rejection samplers that restart execution from the beginning upon observation failure (Figure 4.6b).

Definition 46 (tie_itree)**.** *For* $t : \mathcal{T}^{it}_{\mathbf{1}+\Sigma}$*, define* tie_itree $t : \mathcal{T}^{it}_{\Sigma}$ *as:*

$$\text{tie\_itree}\ t \triangleq \text{ITree.iter}\ (\lambda\_.\ t)\ ().$$

The overall compilation from CF trees to ITrees is then given by the composition of to_itree_open with tie_itree:

Definition 47 (to_itree)**.** *For* $t : \mathcal{T}^{cf}_{\Sigma}$*, define* to_itree $t : \mathcal{T}^{it}_{\Sigma}$ *as:*

$$\text{to\_itree}\ t \triangleq \text{tie\_itree}\ (\text{to\_itree\_open}\ t).$$

Interaction tree semantics We wish to define an analogue itwp of the cwp semantics for ITree samplers and prove the correctness of ITree generation with respect to it, such that itwp f (tie_itree (to_itree_open t)) = tcwp t f for all $t : \mathcal{T}^{cf}_{\Sigma}$ and $f : \Sigma \to \mathbb{R}^{\infty}_{\geq 0}$. This turns out to be awkward, however, due to the lack of induction principle for ITrees.

To overcome the problem, we observe that ITree samplers form an *algebraic CPO* [Gun92, Chapter 5], i.e., a domain in which all elements can be obtained as suprema

of ω-chains of finite approximations. Moreover, the types $\mathbb{R}^{\infty}_{\geq 0}$ of extended reals and $\mathbb{P}$ of propositions are CPOs. We exploit these observations to provide a special kind of induction principle for coinductive trees (see [Gun92, Lemma 5.24]). Such a principle enables the definition of Scott-continuous [Sco70, AJ94] functions like itwp, and gives rise to a powerful suite of proof principles for reasoning about them via reduction to inductive proofs over inductive structures (for which Coq is much better suited than for coinduction). See Chapter 6 for a full exposition of the theory of algebraic coinductives and Chapter 7 (especially Section 7.6) for its application to giving weakest pre-expectation semantics to ITree samplers.

End-to-end compiler The compiler pipeline steps are composed via cpGCL_to_itree and proved correct by Theorem 8 (with positivity constraint on $\mathsf{wlp}_{\mathbf{false}}\ c\ \mathbf{1}\ \sigma$ assuring that the program does not condition on contradictory observations):

Definition 48 (cpGCL_to_itree)**.** *For* c : cpGCL *and* $\sigma : \Sigma$*, define*

$$\mathsf{cpGCL\_to\_itree}\ c\ \sigma \triangleq \mathsf{to\_itree}\ (\mathsf{debias}\ (\mathsf{elim\_choices}\ (\mathsf{compile}\ c\ \sigma))).$$

Theorem 8 (Compiler Correctness)**.** *Let* c : cpGCL, $f : \Sigma \to \mathbb{R}^{\infty}_{\geq 0}$, *and* $\sigma : \Sigma$ *such that* $0 < \mathsf{wlp}_{\mathbf{false}}\ c\ \mathbf{1}\ \sigma$. *Then,*

$$\mathsf{cwp}\ c\ f\ \sigma = \mathsf{itwp}\ f\ (\mathsf{cpGCL\_to\_itree}\ c\ \sigma).$$

Theorem 8 establishes semantics preservation of the compiler pipeline with respect to itwp. However, it doesn't directly guarantee any properties of samples generated by the resulting ITrees. Drawing on basic measure theory [Hal13] and the theory of uniform-distribution modulo 1 [Wey16, KN12, BG22], the next chapter extends the result of Theorem 8 to show that sequences of generated samples are equidistributed with respect to the cwp semantics of their source programs.

5 Correctness of Sampling

I can prove anything by statistics
except the truth.

George Canning

Given a suitable source of i.i.d. randomness (Section 5.1), we say that a sampler for program c : cpGCL is correct if it produces a sequence $x_n : \mathbb{N} \to \Sigma$ such that for any observation $Q : \Sigma \to \mathbb{P}$ over terminal program states, the *proportion of samples* falling within Q asymptotically converges to the expected value of $[Q]$ (i.e., the probability of Q) according to c's cwp semantics. In other words, a sampler is correct when the samples it produces are *equidistributed* [BG22] with respect to cwp.

This section formalizes the notion of equidistribution described above and proves the main sampling equidistribution theorem (Theorem 10, Section 5.3). We first clarify what is meant by "a suitable source of randomness" (Section 5.1) and then re-cast the problem of inference as that of computing a measure (Section 5.2).

5.1 The Source of Randomness

For the source of randomness, we assume access to potentially infinite sequences of uniformly distributed bits from the sampler's operating environment. The space of infinite sequences of bits (i.e., *bitstreams*), denoted $2^{\mathbb{N}}$, is called the *Cantor space* [Kec12]. The Cantor space, equipped with the Borel σ-algebra induced from closure under countable unions and complements of basic sets of of bitstreams defined by finite prefixes (see the following paragraph), and with the uniform Lebesgue measure, forms the standard Borel probability space Ω. We assume the ability to draw elements of $2^{\mathbb{N}}$ at random from Ω, i.e., uniformly at random.

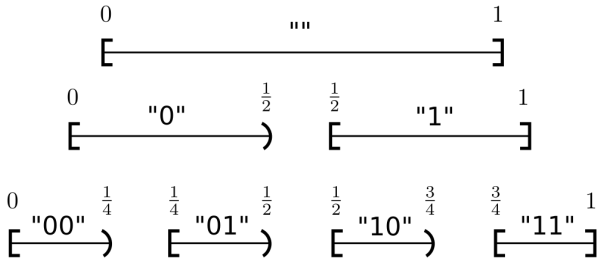
Bisecting the Unit Interval To help visualize the measure on $2^{\mathbb{N}}$, consider the bisection scheme shown in Figure 5.1a for identifying strings of bits (e.g., "0", "01", "011", etc.)

with dyadic subintervals (Definition 49) of the unit interval $[0, 1]$. Let $I(\omega)$ denote the interval corresponding to string ω, and $B(\omega) \subseteq 2^{\mathbb{N}}$ the *basic set* of bitstreams with prefix ω (i.e., $B(\omega) = \{s \mid \omega \sqsubseteq s\}$ where '$\omega \sqsubseteq s$' means that string ω is a finite prefix of bitstream s). We arrange for the *measure* of $B(\omega)$, denoted by $\mu_\Omega(B(\omega))$, to be equal to the *length* of the interval $I(\omega)$ which is equal to 2^{-n} where n is the length of ω. We then define the *source of randomness* Ω to be the measure space obtained by equipping $2^{\mathbb{N}}$ with measure μ_Ω lifted to the Borel σ-algebra Σ_Ω of countable unions and complements of basic sets, coinciding with the standard Lebesgue measure λ on the Borel σ-algebra generated by subintervals of $[0, 1]$.

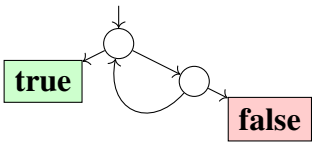
Definition 49 (Dyadic Interval)**.** *A dyadic interval is an interval of real numbers whose endpoints are of the form* $\frac{j}{2^n}$ *and* $\frac{j+1}{2^n}$*, where* j *is an integer and* n *is a natural number.*

Since we are only interested in subintervals of $[0, 1]$, we always have $0 \leq j \leq 2^n$. We say that string ω_a is a prefix of another string ω_b iff there exists an ω' such that $\omega_a \mathbin{+\!\!+} \omega' = \omega_b$ (where '++' denotes string concatenation), written $\omega_a \sqsubseteq \omega_b$). Note that $\omega_a \sqsubseteq \omega_b$ iff $I(\omega_b) \subseteq I(\omega_a)$. Consequently, if neither $\omega_a \sqsubseteq \omega_b$ nor $\omega_b \sqsubseteq \omega_a$ (i.e., if ω_a and ω_b are *incomparable*), then we have $I(\omega_a) \cap I(\omega_b) = \emptyset$, and we say that ω_a and ω_b are *disjoint*.

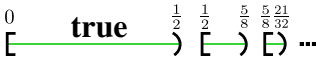
By taking the closure of the collection of all basic sets under unions and complements, we obtain the Borel σ-algebra Σ_Ω on Ω. Since measures on Σ_Ω are uniquely determined by their action on basic sets, to define a measure on the whole of Σ_Ω it suffices to define it just for the basic sets. For us, the measure of basic set $B(\omega)$ (i.e., the length of the interval corresponding to bitstring ω) is readily given as $\dfrac{1}{2^n}$ where n is the length of ω. The overall measure this induces on Ω corresponds to the standard Lebesgue measure on $[0, 1]$, and thus calculation of the measure of a Borel subset of Ω can be understood as calculation of the sum of lengths of disjoint dyadic subintervals of $[0, 1]$.



(a) Bisection scheme identifying bitstrings with dyadic subintervals of [0, 1].



(b) ITree sampler $t_{\frac{2}{3}}$.



(c) Preimage intervals of event {**true**} under $f_{t_{\frac{2}{3}}}$.

Figure 5.1: Interval bisection scheme (left) and its application to ITree $t_{\frac{2}{3}}$ (right).

5.2 Inference as Measure

We view ITree sampler $t : \mathcal{T}_\Sigma^{it}$ as a partial measurable function $f_t : \Omega \rightharpoonup \Sigma$ from Ω to the sample space (Σ, Σ_Σ) where Σ_Σ is the discrete (power set) σ-algebra on the space of program states Σ. Evaluation of f_t on bitstream $s : 2^{\mathbb{N}}$ has two possible outcomes:

1. The sampler *diverges*, consuming bits ad infinitum without ever producing a sample. In that case, we have $f_t(s) = \bot$, i.e., f_t is *undefined* on s. We admit samplers for which such infinite executions are permitted but occur with probability 0 (i.e., the set $D \subseteq 2^{\mathbb{N}}$ of diverging inputs has measure 0). Or,

2. a value x is produced after consuming a finite prefix $\omega \sqsubseteq s$. Thus, the function f_t sends all bitstreams in the basic set $B(\omega)$ to output x.

For example, consider the ITree sampler $t_{\frac{2}{3}}$ in Figure 5.1b yielding **true** with probability $\frac{2}{3}$. The *preimage* $f_{t_{\frac{2}{3}}}^{-1}(\{\textbf{true}\})$ of event {**true**} under $f_{t_{\frac{2}{3}}}$ (i.e., the set of

bitstreams sent by $f_{t_{\frac{2}{3}}}$ to **true**) has measure $\frac{2}{3}$. To see why, observe that $t_{\frac{2}{3}}$ contains infinitely many disjoint paths to **true** ("0", "100", "10100", etc.), with corresponding interval lengths $\frac{1}{2}$, $\frac{1}{8}$, $\frac{1}{32}$, etc., a geometric series with sum $\sum_k^\infty \frac{1}{2} \cdot \frac{1}{4}^k = \frac{\frac{1}{2}}{1-\frac{1}{4}} = \frac{2}{3}$.

We can exploit this observation to let the measure of any event $Q \subseteq \{\textbf{true}, \textbf{false}\}$ be equal to the measure of its preimage under $f_{t_{\frac{2}{3}}}$, thus inducing the following probability measure $\mu_{t_{\frac{2}{3}}} : \{\textbf{true}, \textbf{false}\} \to \mathbb{R}^{\leq 1}_{\geq 0}$ (where $\lambda(I)$ denotes the length of interval I):

$$\begin{array}{llllll}
\mu_{t_{\frac{2}{3}}}(\emptyset) & = & \mu_\Omega(f^{-1}_{t_{\frac{2}{3}}}(\emptyset)) & = & \lambda(\emptyset) & = & 0 \\
\mu_{t_{\frac{2}{3}}}(\{\textbf{true}\}) & = & \mu_\Omega(f^{-1}_{t_{\frac{2}{3}}}(\{\textbf{true}\})) & = & \lambda([0, \frac{2}{3})) & = & \frac{2}{3} \\
\mu_{t_{\frac{2}{3}}}(\{\textbf{false}\}) & = & \mu_\Omega(f^{-1}_{t_{\frac{2}{3}}}(\{\textbf{false}\})) & = & \lambda([\frac{2}{3}, 1]) & = & \frac{1}{3} \\
\mu_{t_{\frac{2}{3}}}(\{\textbf{true}, \textbf{false}\}) & = & \mu_\Omega(f^{-1}_{t_{\frac{2}{3}}}(\{\textbf{true}, \textbf{false}\})) & = & \lambda([0, 1]) & = & 1
\end{array}$$

Computing Preimages We can generalize the above method to induce a probability measure $\mu_t : \Sigma \to \mathbb{R}^{\leq 1}_{\geq 0}$ from any $t : \mathcal{T}^{it}_\Sigma$ by letting $\mu_t(Q) = \mu_\Omega(f_t^{-1}(Q))$ (the *pushforward measure* of Q under f_t), assigning to any event $Q : \Sigma \to \mathbb{P}$ a probability equal to the measure in Ω of its preimage under f_t. However, the preimage $f_t^{-1}(Q)$ is not easy to determine in general, as it may be the union of infinitely many small intervals scattered throughout $[0, 1]$ in a complicated arrangement depending on the structure of t.

The formal construction of $f_t^{-1}(Q)$ is deferred to Section 7.3, as it depends on the tools of the AlgCo framework developed in Chapter 6. Still, we may hope at this point to be able to choose a representation for sets of bitstrings. Since languages over finite alphabets are always countable, one possibility is the type $\mathbb{N} \to$ option (list bool) of nat-indexed sequences of optional bitstrings. Indeed, that was the original choice of representation for earlier versions of this work, and it can be shown to possess a semiring structure which allows for powerful equational reasoning (up to the appropriate

equivalence relation). However, we find a coinductive tree-based encoding (see Section 7.3) of sets to be a more natural fit within the framework of AlgCo and so favor it over the nat-indexed representation. Since this type is used to encode an analogue of the class Σ^1_0 of sets (see Section 2.8), we refer to it as 'Sigma01'.

5.3 Equidistribution

To prove correctness of samplers generated by Zar, we show that any sequence of samples produced by them is *equidistributed* with respect to the cwp semantics of the input programs. Our strategy is to assume uniform distribution of the source of randomness Ω, and "push it forward" through the sampler to obtain the desired result. This section builds on the theory of uniform distribution modulo 1 – in particular, the class Σ^0_1 of countable unions of rational bounded intervals – adapted to collections of bitstreams.

Uniform distribution of Ω We assume access to a uniformly distributed unending sequence of bitstreams. But what does it mean for a sequence of bitstreams to be uniformly distributed? We cannot simply assert that any two bitstreams occur with equal probability, since any particular bitstream may have probability zero, even for nonuniform distributions. Instead, we turn to a variation of the classic notion of "uniform distribution modulo 1" [KN12], generalized to the class Σ^0_1 of subsets of $2^{\mathbb{N}}$.

A subset $U \subseteq \mathcal{P}(2^{\mathbb{N}})$ is said to be Σ^0_1 when it is equal to $\bigcup_i^\infty B(\omega_i)$ for some countable collection $\{B(\omega_i)\}$ of basic sets. We remark that $f_t^{-1}(Q)$ is Σ^0_1 for all $Q : \Sigma \to \mathbb{P}$ and $t : \mathcal{T}^{it}_\Sigma$. The required notion of uniform distribution now follows:

Definition 50 (Σ^0_1-u.d.)**.** *A sequence $\{x_i\}$ of bitstreams is Σ^0_1-uniformly distributed (Σ^0_1-u.d.) when for every $U : \Sigma^0_1$,*

$$\lim_{n\to\infty} \frac{1}{n} \sum_{i=0}^{n-1} [x_i \in U] = \mu_\Omega(U).$$

Sampling correctness for interaction tree samplers can then be stated by the following equidistribution theorem (recall that f_t denotes the interpretation of ITree t as a measurable function $f_t : \Omega \rightharpoonup \Sigma$):

Theorem 9 (ITree equidistribution)**.** *Let A be a type, $t : \mathcal{T}_A^{it}$ be an interaction tree sampler over A, $\sigma : \Sigma$ be a program state, $\{x_n\}$ be a Σ_1^0-u.d. sequence of bitstreams, and $Q : \Sigma \to \mathbb{P}$ be a predicate over program states, Then, the sequence $\{f_t(x_n)\}$ of samples is* itwp*-equidistributed with respect to t:*

$$\lim_{n\to\infty} \frac{1}{n} \sum_{i=0}^{n-1} [Q\,(f_t(x_i))] = \mathsf{itwp}\ t\ [Q].$$

Theorem 9 is proved by reduction to an analogous theorem on cotrees (Theorem 23), the details of which are given in Section 7.4 using the tools of the AlgCo framework. The basic idea is to show that itwp t $[Q]$ is equal to $\mu_\Omega(f_t^{-1}(Q))$ (the measure of the preimage of Q under f_t). The goal then follows immediately by our assumption that $\{x_n\}$ is Σ_1^0-u.d. (by letting $U = f_t^{-1}(Q)$ in Definition 118).

The ITree equidistribution theorem assumes that every bitstream x_n produces a corresponding sample $f_t(x_n)$, i.e., that the sampler terminates on every x_n. This is a subtle point, because earlier we assumed only that the program t was compiled from terminates *with probability* 1, not necessarily for every possible bitstream (see Section 7.3 for related discussion). However, since the set of bitstreams for which such a sampler diverges has measure 0, the probability of any particular bitstream generated uniformly at random belonging to that set is 0. It follows that the probability of a divergent bitstream occurring in $\{x_n\}$ is 0 and so our equidistribution theorem *holds with probability* 1 for any t that terminates with probability 1.

Finally, the equidistribution result can be lifted to samplers compiled from cpGCL programs through Theorems 3 and 8.

Theorem 10 (cpGCL equidistribution)**.** *Let* $c : \mathsf{cpGCL}$ *be a command,* $\sigma : \Sigma$ *a program state,* $\{x_n\}$ *a* Σ_1^0*-u.d. sequence of bitstreams,* $Q : \Sigma \to \mathbb{P}$ *a predicate over program states, and* $t : \mathcal{T}_\Sigma^{it} = \mathsf{cpGCL\_to\_itree}\ \mathsf{c}\ \sigma$ *the ITree sampler compiled from c. Then, the sequence* $\{f_t(x_n)\}$ *of samples is* cwp*-equidistributed with respect to c:*

$$\lim_{n\to\infty} \frac{1}{n} \sum_{i=0}^{n-1} [Q\ (f_t(x_i))] = \mathsf{cwp}\ c\ [Q]\ \sigma.$$

The proof of Theorem 10 makes substantial use of the AlgCo framework, so we defer its details to Chapter 7 after having introduced the basic concepts of algebraic coinductives in Chapter 6.

6 Algebraic Coinductives

Natura non facit saltus.

Leibniz

This chapter presents AlgCo (Algebraic Coinductives), a practical framework for inductive reasoning over coinductive types such as conats, streams, and infinitary trees with finite branching factor. The key idea is to exploit the notion of *algebraic CPO* from domain theory to define continuous operations over coinductive types via primitive recursion on "dense" collections of their elements, enabling a convenient strategy for reasoning about algebraic coinductives by straightforward proofs by induction. We implement the AlgCo library in Coq and demonstrate its utility by verifying a stream variant of the sieve of Eratosthenes, a regular expression library based on coinductive tries, and weakest pre-expectation semantics for potentially nonterminating sampling processes over discrete probability distributions in the random bit model.

We begin by motivating the need for a canonical elimination principle for coinductive types (Section 6.1). We then provide introductions to the basic notions of algebraic CPOs (Section 6.2), (co-)continuous extensions (Sections 6.3 and 6.4), and (co-)continuous properties (Section 6.5). Finally, we give concrete examples of algebraic CPOs and their applications to conats (Section 6.6), streams (Section 6.7), and infinitary tries (Section 6.8). The primary application of the AlgCo framework to coinductive binary trees is described in detail in Chapter 7.

6.1 Appetite for Elimination

As a tool for defining and proving correctness of computations over well-founded data, the principle of induction is intimately familiar to most computer scientists. Consequently, proof assistants like Coq and Agda provide ergonomic support for

programming and proving with induction [INR22, Tea22]. The dual principle of *coinduction* [KS17], on the other hand, is not nearly as well supported.

Coinduction provides a natural means for programming with and verifying properties of infinitary structures such as conats (natural numbers extended with a "point at infinity"), streams (infinite lists) [Chl22], and potentially nonterminating decision processes such as samplers for discrete distributions compiled from probabilistic programs (Section 4.5). However, coinductive reasoning has a tendency to betray intuition, and proof assistants like Coq and Agda are designed with a noticeable bias toward induction, which can exacerbate the inherently unintuitive nature of coinduction. As a result, the use of coinduction in a proof assistant is often plagued by technical snags due to rigid syntactic guardedness conditions (limiting the range of allowable coinductive definitions), and the generation of coinduction hypotheses that interact poorly with automation tactics [HNDV13, Chl22]). Hur et al. proposed – and Pous later generalized [Pou16] – parameterized coinduction (paco [HNDV13]) as a way to address rigid syntactic checks through the use of a semantic notion of guardedness. While paco substantially upgrades coinduction in Coq, it does not represent a fundamental departure from primitive coinduction.

We propose an alternative strategy: instead of generalizing coinduction in Coq (e.g., as paco does to semantic guardedness), we consider a subset of coinductive types – those corresponding to algebraic CPOs [Jun90, Gun92] – for which a completely different strategy can be applied for reasoning about coinductive programs that is more suited to the inherently inductive disposition of proof assistants like Coq.

To illustrate our approach, consider the problem of defining a filter operation for coinductive streams (i.e., infinite lists). In Haskell, it is possible to filter a stream by a given predicate – e.g., the stream of even numbers is given by $[n \mid n \leftarrow [0..], n \text{ `}mod\text{`} 2 = 0]$. We can attempt to implement a similar filter operation for streams in Coq as follows:

```
CoFixpoint filter (P : ℕ → 𝔹) (s : Stream ℕ) ≜
  match s with Cons n s' ⇒
    if P n then Cons n (filter P s') else filter P s'
  end.
```

This definition is rejected, however, due to presence of an unguarded recursive call (not wrapped in a **Cons** constructor) in the 'else' branch. Indeed, it is a good thing that it is rejected, or else we could create a divergent term (rendering the logic of Coq inconsistent) by filtering any stream by $P := \lambda \_.$ **false**. While there may be specific circumstances in which we could prove it safe to filter a stream by a given predicate (i.e., when the resulting stream will be "productive" [Chl22]), we cannot do that because filter is not definable in the first place.

A common solution (taken, e.g., by Xia et al. in the interaction tree library [XZH+20]) is to add a constructor to the stream type for so-called "silent steps" (*Tau* nodes). Tau nodes trivially satisfy the guardedness checker – wrap unguarded co-recursive calls by applications of Tau – but lead to extra cases in definitions and proofs, and to unnecessary execution overhead. Moreover, replacing points of divergence with infinite sequences of Taus passes the responsibility of handling divergence to consumers of the stream, leading to complications in subsequent computation and analysis.

As an example of a complication of Tau, consider taking the infinite sum of a stream of reals in which Tau nodes can appear. Lacking a general induction principle for streams, we resort to a coinductive relation between streams and their sums (where $\mathbb{R}^{\infty}_{\geq 0}$ denotes the nonnegative reals extended with $+\infty$, necessary in case of divergent series):

```
CoInductive sum : Stream ℝ∞≥0 → ℝ∞≥0 → ℙ ≜
| sum_tau : ∀ s r, sum s r → sum (Tau s) r
| sum_cons : ∀ s x r, sum s r → sum (Cons x s) (x + r).
```

and attempt to prove that the relation is functional:

```
Lemma sum_functional (s : Stream ℝ∞≥0) (a b : ℝ∞≥0) :
  sum s a → sum s b → a = b.
Proof. ... Abort. (* unprovable *)
```

But sum_functional is unprovable because it isn't true. The problem is that coinductive relations are interpreted as the *greatest* relations closed under their rules, and so tend to relate more pairs of elements than intended. E.g., the stream $\Omega \triangleq$ Tau Ω is related by sum to every $r : \mathbb{R}$, so sum is clearly not a functional (i.e., deterministic) relation because, e.g., sum Ω 0 and sum Ω 1, but $0 \neq 1$). We could constrain the lemma to apply only to streams containing no infinite sequences of Taus, but then we take on the burden of proving this side condition for all of our stream constructions. Defining the sum relation via paco does not help because it suffers from the same fundamental problem, being defined as the *greatest* fixed point of a monotone functional.

We take an alternative approach (similar to that in [RN22]) for defining sum inspired by domain theory. We first define an inductive analogue of sum on lists:

```
Fixpoint lsum (l : List ℝ∞≥0) : ℝ∞≥0 ≜
  match l with
  | nil ⇒ 0
  | cons x l' ⇒ x + lsum l'
  end.
```

and then for s : Stream $\mathbb{R}^{\infty}_{\geq 0}$ we define sum $s : \mathbb{R}^{\infty}_{\geq 0} \triangleq$ sup {lsum l | l is a finite prefix of s}. This definition of sum maps every stream to a unique element of $\mathbb{R}^{\infty}_{\geq 0}$, even in the presence of Tau nodes. We may ask: under what conditions is it possible to define functional mappings on coinductives in this way? For the answer we turn to a fundamental result of domain theory (Lemma 5):

- $\mathbb{R}^{\infty}_{\geq 0}$ is a CPO (complete partial order),
- Stream $\mathbb{R}^{\infty}_{\geq 0}$ is an *algebraic* CPO,
- and nsum is *monotone*.

We call sum the *continuous extension* of lsum. It is continuous by construction, and many proofs about it (specifically, proofs of *continuous properties*, see Section 6.5) can be reduced to straightforward proofs by induction over lists. Moreover, *every continuous function* from streams into $\mathbb{R}^{\infty}_{\geq 0}$ (or indeed, from any algebraic CPO into another CPO) can be obtained in this way, that is, as the continuous extension of a monotone function over finite elements.

A basic connection between coinductive types and CPOs that can be understood as follows: A coinductive type, interpreted as the final coalgebra of a given functor [Hag89], is equipped (by finality) with a canonical introduction principle for defining elements of the type. The completeness property of a CPO can likewise be viewed as a kind of introduction principle for which elements of the CPO are introduced as suprema of directed sets of approximations. Thus coinductive types naturally form CPOs (e.g., the type of streams can be seen as the "completion" of the type of lists wrt. suprema of directed sets [Adá21]). Furthermore, by exploiting algebraicity (the existence of a dense compact subset) of the domain and completeness of the codomain (in this case Stream $\mathbb{R}^{\infty}_{\geq 0}$ and $\mathbb{R}^{\infty}_{\geq 0}$, respectively), we are able to provide a *continuous elimination principle* for algebraic domains (Lemma 5) for defining continuous functions like filter and sum as well as continuous predicates and relations (Section 6.5) over them.

In this chapter we elucidate the power and generality of the concept of algebraic CPOs by providing a comprehensive framework (*AlgCo*, short for *Algebraic Coinductives*) for programming and proving with continuous functions in Coq over the class of coinductive types forming algebraic CPOs (including commonly used structures such as conats, streams, and infinitary trees), and demonstrate its utility on a handful of interesting use cases including the semantics of probabilistic programming languages (Chapter 7). In Section 6.7.1, we show that the aforementioned filter operation can be defined with the tools of AlgCo without the need for Tau nodes at all.

Limitations of AlgCo Continuous extensions such as $\mathsf{sum} : \mathsf{Stream}\ \mathbb{R}^{\infty}_{\geq 0} \to \mathbb{R}^{\infty}_{\geq 0}$ described above are defined via a nonconstructive supremum operator (see Section 2.10) and thus are not computable in general. However, we provide Haskell extraction primitives for some special cases (e.g., lazy coiteration (Section 6.6.2) and cofolds (Section 6.7.1)) that suffice to cover a wide range of practically useful operations. See Section 6.7.5 for discussion on the issue of computability of continuous extensions.

The techniques described in this chapter apply only to types which form algebraic CPOs. While this includes many useful coinductive types such as streams and infinitary binary trees, there exist many interesting types for which it does not apply, e.g., trees that are infinite in both depth *and breadth*.

Computing Suprema Although suprema of directed sets are not generally computable, we can use the axioms of indefinite description and excluded middle [Cha17] to define a sup (inf) operator (special cases of Hilbert's epsilon operator [coq23a]) allowing succinct expression of suprema (infima) in a classical style within computational contexts in Coq [Cha10]. We define $\mathsf{sup} : (\mathbb{N} \to A) \to A$ such that $\mathsf{sup}\ f$ is the supremum of any directed $f : \mathbb{N} \to A$, and $\mathsf{inf} : (\mathbb{N} \to A) \to A$ such that $\mathsf{inf}\ f$ is the infimum of any downward-directed $f : \mathbb{N} \to A$. Any attempt to compute with these operators will become blocked [Ler15], but in some special cases, they can be implemented by extraction primitives for lazy execution in Haskell (see Sections 3.2 and 6.7.5 for discussion).

6.2 Algebraic CPOs

A CPO is traditionally said to be *algebraic* when it contains a subset of "basis" elements that can be used to approximate any element of the domain [Gun92]. These basis elements are required to be *compact*:

Definition 51 (Compact element)**.** *Let A be an ordered type. An element $x : A$ is compact when for every directed collection $U : \mathbb{N} \to A$ such that $\mathsf{sup}\ U = x$, there exists an $i : \mathbb{N}$ such that $U\ i = x$.*

An element x is compact when it can only be trivially approximated, i.e., when any directed set that "converges" to x must contain x itself. Elements of many data types (including binary trees) are compact precisely when they are *finite* (but this is not always the case, see, e.g., [Gun92, Section 5.1]), thus it is usually reasonable to think of compactness as simply an alternate definition of finiteness providing a more "extensional" characterization of what it means to be finite.

Although compact elements are not necessarily finite, they *are* finitely approximable, i.e., any directed set "converging" to a compact element x contains a finite subset that also converges to x (see Section 6.8 for an algebraic CPO whose basis elements are compact but not finite). We say that a type A is compact when every $x : A$ is compact. Compactness is essential for us because it coincides with the presence of an induction principle; a key idea of algebraic coinductives is to reduce reasoning about continuous functions over coinductive types to purely inductive reasoning over basis elements (see, e.g., Lemmas 13, 14, 23, 24 and Theorems 16, 17).

Definition 52 (Compact type)**.** *An ordered type A is compact when x is compact for every $x : A$.*

An essential fact about compact types is that all monotone functions on them are trivially continuous:

Lemma 4 (Monotone functions on compact types are continuous)**.** *Let A be a compact ordered type, B an ordered type, and $f : A \to B$ a monotone function. Then, f is continuous.*

We adapt the traditional definition of algebraicity to a type-theoretic framework by saying that a CPO A is algebraic when there exists a basis type B that can be injected into A (hence is like a subset of A) and is "dense" in A, i.e., all elements of A can be obtained as suprema of directed collections of elements of B injected into A. Think by analogy of the rational and real numbers: the rationals are finitely representable and thus compact, and any real number can be obtained as the supremum of a collection of rationals injected into the reals (e.g., $\sqrt{2} = \mathsf{sup}\,\{x : \mathbb{Q} \mid x^2 < 2\}$).

Definition 53 (Dense)**.** *Let A be an ω-CPO and B an ordered type. B is dense in A when there exist continuous operations*

$$\mathsf{incl}_{B,A} : B \to A$$

$$\mathsf{idl}_{B,A} : A \to \mathbb{N} \to B$$

such that for all $a : A$,

$$\mathsf{idl}_{B,A}\ a \text{ is an } \omega\text{-chain, and} \qquad \mathsf{sup}\,(\mathsf{incl}_{B,A} \circ \mathsf{idl}_{B,A}\ a) = a.$$

The idl operator (read "ideal") applied to element $x : A$ produces an ω-chain of basis elements whose injections into A converge to x. Strictly speaking, we should only require idl to produce directed sets, but we choose to constrain to ω-chains in hopes of developing a useful notion of general computability of continuous extensions in the future. We omit the subscripts on idl and incl when they are clear from context.

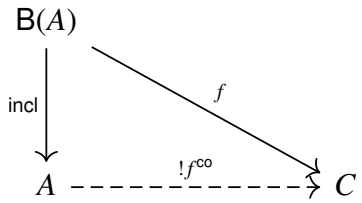
Definition 54 (Algebraic CPO)**.** *Let A be an ω-CPO and B an ordered type. A is an algebraic CPO (aCPO) with basis B when B is compact and dense in A.*

We let $\mathsf{B}(A)$ denote the basis of algebraic CPO A. Some authors (e.g., [Sco70, AJ94]) do not require the basis to be compact (reserving the name *algebraic CPO* for when it is). Algebraic CPOs are closed under the formation of products and sums. The type $A \to B$ is an algebraic CPO when A is finite and B is an algebraic CPO.

6.3 Continuous Extensions

A key idea of the AlgCo framework is to define continuous functions on algebraic CPOs as *continuous extensions* of simpler monotone functions on basis elements, formally characterized by the following lemma (based on [Gun92, Lemma 5.24]):

Lemma 5 (Continuous extension)**.** *Let A be an algebraic CPO, C a CPO, and $f : \mathsf{B}(A) \to C$ a monotone function. Then, there exists a unique continuous function $f^{\mathsf{co}} : A \to C$ (the continuous extension of basis function f) such that $f^{\mathsf{co}} \circ \mathsf{incl} = f$. I.e., the following diagram commutes:*



The main aspects of Lemma 5 are two-fold:

Existence. Any monotone function $f : \mathsf{B}(A) \to C$ defined on the basis of an algebraic CPO can be extended to a continuous function $f^{\mathsf{co}} : A \to C$ on the whole domain A.

Uniqueness. The extension is *unique*. Any continuous function completing the diagram must be equal to f^{co}.

The existence of f^{co} constitutes a *continuous elimination scheme* for algebraic CPOs: to define a continuous function on algebraic CPO A, it suffices instead to define a monotone basis function $f : \mathsf{B}(A) \to C$ (typically by induction, (see Section 6.7 and Chapter 7)) and extend it to $f^{\mathsf{co}} : A \to C$.

Moreover, uniqueness of f^{co} implies that *every* continuous function $g : A \to C$ can be represented as a continuous extension $f^{\mathsf{co}} : A \to C$ for some monotone basis function $f : \mathsf{B}(A) \to C$, namely $f = g \circ \mathsf{incl}$ (see Corollary 1).

Lemma 5 tells us that the continuous functions on algebraic CPOs are precisely the functions that can be "easily" defined (as extensions of basis functions), and gives rise to a

collection of powerful proof principles (e.g., Lemma 6 and Theorem 14) for reasoning about them.

6.3.1 Existence and Uniqueness of Continuous Extensions

We divide the proof of Lemma 5 into two separate theorems for the *existence* and *uniqueness* of continuous extensions. Existence yields the *definitional* principle for continuous extensions:

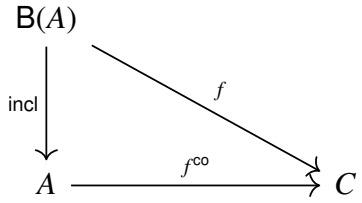
Theorem 11 (Existence of continuous extensions)**.** *Let A be an algebraic CPO, C any CPO, and $f : \mathsf{B}(A) \to C$ a monotone function. Define*

$$f^{\mathsf{co}}\ a \triangleq \mathsf{sup}\ (f \circ \mathsf{idl}\ a).$$

Then,

$$f^{\mathsf{co}} \circ \mathsf{incl} = f.$$

I.e., the following diagram commutes:

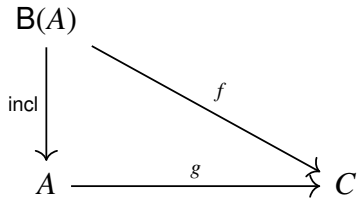


And uniqueness of continuous extensions yields the fundamental *proof* principle:

Theorem 12 (Uniqueness of continuous extensions)**.** *Let A be an algebraic CPO, C any CPO, $f : \mathsf{B}(A) \to C$ a monotone function on the basis of A, and $g : A \to C$ a continuous function on A such that:*

$$g \circ \mathsf{incl} = f,$$

i.e., that the following diagram commutes:



Then, $g = f^{\mathsf{co}}$.

6.3.2 Proof Principles for Continuous Extensions

Theorem 12 immediately yields a useful proof principle: to show that continuous functions f and g are equal, it suffices to show that they are both solutions for x to the equation '$x \circ \mathsf{incl} = f$'. Another useful corollary of Theorem 12 is the following principle for proving equality of continuous extensions via reduction to equality on basis elements.

Lemma 6 (Equivalence of continuous extensions)**.** *Let A be an algebraic CPO, C any CPO, and $f : \mathsf{B}(A) \to C$ and $g : \mathsf{B}(A) \to C$ monotone functions on the basis of A. Then,*

$$f = g \Rightarrow f^{\mathsf{co}} = g^{\mathsf{co}}.$$

Lemma 6 is extremely useful in practice: to show $f^{\mathsf{co}}\ a = g^{\mathsf{co}}\ a$ for all $a : A$, it suffices to show $f\ b = g\ b$ for all $b : \mathsf{B}(A)$, which can often be proved by straightforward induction.

The following generalized point-wise form of the equivalence principle is more powerful than Lemma 6 because it remembers that the basis elements in the antecedent are approximations of the elements appearing in the goal, which may be important when the equality being proved is contingent on some precondition.

Lemma 7 (Generalized equivalence of continuous extensions)**.** *Let A and B be algebraic CPOs, C any CPO, $f : \mathsf{B}(A) \to C$, $g : \mathsf{B}(B) \to C$ monotone functions, $a : A$ and $b : B$. Then,*

$$\forall\, i : \mathbb{N},\ f\ (\mathsf{idl}\ a\ i) = g\ (\mathsf{idl}\ b\ i) \Rightarrow f^{\mathsf{co}}\ a = g^{\mathsf{co}}\ b.$$

We also have the eponymous fact that continuous extensions are continuous:

Theorem 13 (Continuous extensions are continuous)**.** *Let A be an algebraic CPO, C any CPO, and $f : \mathsf{B}(A) \to C$ a monotone function. Then, f^{co} is continuous.*

While an obvious result, the usefulness of Theorem 13 should not be understated. Ad hoc proofs of continuity are often difficult and time consuming. By defining our functions as continuous extensions, we obtain proofs of their continuity for free!

The following corollary, a fairly direct consequence of Theorem 12, shows that every continuous function $g : A \to C$ can be expressed as the continuous extension $(g \circ \mathsf{incl})^{\mathsf{co}}$.

Corollary 1 (Representation of continuous functions)**.** *Let A be an algebraic CPO, C any CPO, and $g : A \to C$ a continuous function. Then,*

$$g = (g \circ \mathsf{incl})^{\mathsf{co}}.$$

Fusion It is well-known from Calculus that continuous real-valued functions are closed under composition. Likewise, the following *fusion law* for continuous extensions shows that any composition of the form $g \circ f^{\mathsf{co}}$ where g is continuous is equal to the continuous extension $(g \circ f)^{\mathsf{co}}$.

Theorem 14 (Fusion)**.** *Let A be an algebraic CPO, B and C CPOs, $f : \mathsf{B}(A) \to B$ a monotone function, and $g : B \to C$ a continuous function. Then,*

$$g \circ f^{\mathsf{co}} = (g \circ f)^{\mathsf{co}}.$$

Furthermore, since continuous extensions are continuous (Theorem 13), we have as a special case for all monotone $f : \mathsf{B}(A) \to B$ and $g : \mathsf{B}(B) \to C$:

$$g^{\mathsf{co}} \circ f^{\mathsf{co}} = (g^{\mathsf{co}} \circ f)^{\mathsf{co}}.$$

A further consequence of the fusion law is that any chain of continuous compositions can be written as a single continuous extension so long as the rightmost (the one directly receiving the input) in the chain is a continuous extension:

$$g_n \circ \ldots \circ g_0 \circ f^{\mathsf{co}} = (g_n \circ \ldots \circ g_0 \circ f)^{\mathsf{co}}$$

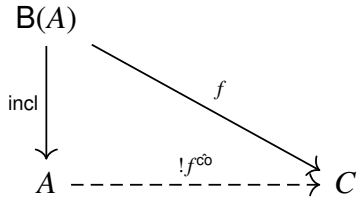
which, by Corollary 1, can be generalized to arbitrary continuous functions:

$$g_n \circ \ldots \circ g_0 = (g_n \circ \ldots \circ g_0 \circ \mathsf{incl})^{\mathsf{co}}.$$

6.4 Cocontinuous Extensions

We can also obtain *cocontinuous* functions (Definition 14) over an algebraic CPO as cocontinuous extensions of *antimonotone* basis functions. Every proof principle for continuous extensions has an analogue for cocontinuous extensions. We list only the essential lemma here:

Lemma 8 (Cocontinuous extension)**.** *Let A be an algebraic CPO, C any lCPO, and $f : \mathsf{B}(A) \to C$ an antimonotone function. Then, there exists a unique cocontinuous function $f^{\hat{\mathsf{co}}} : A \to C$ (the cocontinuous extension of basis function f) such that $f^{\hat{\mathsf{co}}} \circ \mathsf{incl} = f$. I.e., the following diagram commutes:*



Gallery of Fusion Cocontinuous extensions entail a handful of additional fusion laws to complement Theorem 14:

Theorem 15 (Fusion cont'd)**.** *Let A be an algebraic CPO, $f : \mathsf{B}(A) \to B$, and $g : B \to C$.*

For lCPOs B and C, antimonotone f, and l-continuous g,

$$g \circ f^{\hat{\mathsf{co}}} = (g \circ f)^{\hat{\mathsf{co}}}.$$

For CPO B, lCPO C, monotone f, and cocontinuous g,

$$g \circ f^{\mathsf{co}} = (g \circ f)^{\hat{\mathsf{co}}}.$$

For lCPO B, CPO C, antimonotone f, and l-cocontinuous g,

$$g \circ f^{\hat{\mathsf{co}}} = (g \circ f)^{\mathsf{co}}.$$

6.5 (Co-)continuous Properties

Continuous functions with codomain $\mathbb{P}$ are called *continuous properties*. Continuous properties on an algebraic CPO A can be defined as continuous extensions of *monotone properties* on $\mathsf{B}(A)$. By specializing the definition of monotonicity to algebraic CPO A and codomain $\mathbb{P}$, we see that a property $P : \mathsf{B}(A) \to \mathbb{P}$ is monotone when:

$$x \sqsubseteq y \Rightarrow P\,x \Rightarrow P\,y.$$

I.e., P is monotone when its holding on some approximation x implies its holding on all further refinements of x (all y such that $x \sqsubseteq y$). What then does it mean for a property to be continuous? Recalling that a function $f : A \to B$ is ω-continuous when for every ω-chain $C : \mathbb{N} \to A$,

$$f\,(\mathsf{sup}\,C) \simeq \mathsf{sup}\,(f \circ C)$$

specialized to $P^{\mathsf{co}} : A \to \mathbb{P}$:

$$P^{\mathsf{co}}\,(\mathsf{sup}\,C) \iff \mathsf{sup}\,(P^{\mathsf{co}} \circ C)$$

or, equivalently (by Definition 53 and Remark 1):

$$P^{\mathsf{co}}\,a \iff \exists i.\, P\,(\mathsf{idl}\,a\,i).$$

We see that the property P^{co} holds on element $a : A$ if and only if its restriction to $\mathsf{B}(A)$ holds for *some* approximation of a. This leads us to the definition of continuous property:

Definition 55 (Continuous property)**.** *Let A be an algebraic CPO. A predicate $P : A \to \mathbb{P}$ is a continuous property on A when for all $a : A$:*

$$P\ a \iff \exists i.\ P\ (\mathsf{incl}\ (\mathsf{idl}\ a\ i)).$$

We remark that by Corollary 1, every continuous property on algebraic CPO A can be expressed as a continuous extension of the form P^{co} for some monotone $P : \mathsf{B}(A) \to \mathbb{P}$. We also remark that continuous extensions of decidable properties are semi-decidable.

We derive introduction and elimination principles for continuous extensions of monotone $P : \mathsf{B}(A) \to \mathbb{P}$:

$$\textsc{co-intro}\ \frac{P\ (\mathsf{idl}\ a\ i)}{P^{\mathsf{co}}\ a} \qquad \textsc{co-elim}\ \frac{P^{\mathsf{co}}\ a}{\exists i.P\ (\mathsf{idl}\ a\ i).}$$

The dual notion of *cocontinuous property* now follows:

Definition 56 (Cocontinuous property)**.** *Let A be an algebraic CPO. A predicate $P : A \to \mathbb{P}$ is a cocontinuous property on A when for all $a : A$:*

$$P\ a \iff \forall i.\ P\ (\mathsf{incl}\ (\mathsf{idl}\ a\ i)).$$

I.e., a cocontinuous property $P : A \to \mathbb{P}$ holds on element $a : A$ if and only iff its restriction to $\mathsf{B}(A)$ holds for *all* approximations of a. We derive introduction and elimination principles for cocontinuous extensions of antimonotone $P : \mathsf{B}(A) \to \mathbb{P}$:

$$\textit{ĉo}\textsc{-intro}\ \frac{\forall i, P\ (\mathsf{idl}\ a\ i)}{P^{\hat{\mathsf{co}}}\ a} \qquad \textit{ĉo}\textsc{-elim}\ \frac{P^{\hat{\mathsf{co}}}\ a}{P\ (\mathsf{idl}\ a\ i).}$$

6.6 Conats

Our first concrete example of a coinductive algebraic CPO is the type $\mathbb{N}^{\mathsf{co}}$ of *conats*, the natural numbers extended with a "point at infinity" $\omega_{\mathbb{N}}$.

Definition 57 ($\mathbb{N}^{\mathsf{co}}$ (conat))**.** *Define the type* $\mathbb{N}^{\mathsf{co}}$ *of conats coinductively by the formation rules:*

$$\frac{}{\mathbf{cozero} : \mathbb{N}^{\mathsf{co}}}\ \textsc{conat-zero} \qquad \frac{n : \mathbb{N}^{\mathsf{co}}}{\mathbf{cosucc}\ n : \mathbb{N}^{\mathsf{co}}}\ \textsc{conat-succ}$$

Definition 58 ($\omega_{\mathbb{N}}$)**.** *Define the infinite conat* $\omega_{\mathbb{N}}$ *by coinduction:*

$$\omega_{\mathbb{N}} \triangleq \mathbf{cosucc}\ \omega_{\mathbb{N}}.$$

The order relation on $\mathbb{N}^{\mathsf{co}}$ is the usual ordering of natural numbers extended so that $n \sqsubseteq \omega_{\mathbb{N}}$ for all $n : \mathbb{N}^{\mathsf{co}}$.

Remark 4 ($\mathbb{N}^{\mathsf{co}}$ is a CPO). The inductive type $\mathbb{N}$ is not a CPO because there exist directed sets of natural numbers (e.g., $\mathbb{N}$ itself) which have no upper bound and thus no supremum. Conats, on the other hand, have suprema for all sets, because $n \sqsubseteq \omega_{\mathbb{N}}$ for all $n : \mathbb{N}^{\mathsf{co}}$.

The type of $\mathbb{N}$ natural numbers serves as a compact basis for $\mathbb{N}^{\mathsf{co}}$, where the inclusion map $\mathsf{incl}_{\mathbb{N},\mathbb{N}^{\mathsf{co}}} : \mathbb{N} \to \mathbb{N}^{\mathsf{co}}$ injects natural numbers into $\mathbb{N}^{\mathsf{co}}$, and $\mathsf{idl}_{\mathbb{N},\mathbb{N}^{\mathsf{co}}} : \mathbb{N}^{\mathsf{co}} \to \mathbb{N} \to \mathbb{N}$ generates convergent chains of finite approximations of conats.

Definition 59 ($\mathsf{incl}_{\mathbb{N},\mathbb{N}^{\mathsf{co}}}$)**.** *For* $n : \mathbb{N}$*, define* $\mathsf{incl}_{\mathbb{N},\mathbb{N}^{\mathsf{co}}}\ n : \mathbb{N}^{\mathsf{co}}$ *by induction on* n*:*

$\mathsf{incl}_{\mathbb{N},\mathbb{N}^{\mathsf{co}}} : \mathbb{N} \to \mathbb{N}^{\mathsf{co}}$

$\mathbf{O} \triangleq \mathbf{cozero}$

$\mathbf{S}\ n \triangleq \mathbf{cosucc}\ (\mathsf{incl}_{\mathbb{N},\mathbb{N}^{\mathsf{co}}}\ n)$

Definition 60 ($\mathsf{idl}_{\mathbb{N},\mathbb{N}^{\mathrm{co}}}$)**.** *For* $n : \mathbb{N}^{\mathrm{co}}$*, and* $i : \mathbb{N}$*, define* $\mathsf{idl}_{\mathbb{N},\mathbb{N}^{\mathrm{co}}}\ n\ i : \mathbb{N}$ *by induction on* i*:*

$$\mathsf{idl}_{\mathbb{N},\mathbb{N}^{\mathrm{co}}} : \mathbb{N}^{\mathrm{co}} \to \mathbb{N} \to \mathbb{N}^{\mathrm{co}}$$

$$\mathbf{cozero} \quad \_ \quad \triangleq \quad \mathbf{O}$$

$$\_ \quad \mathbf{O} \quad \triangleq \quad \mathbf{O}$$

$$(\mathbf{cosucc}\ n)\ \ (\mathbf{S}\ i) \quad \triangleq \quad \mathbf{S}\ (\mathsf{idl}_{\mathbb{N},\mathbb{N}^{\mathrm{co}}}\ n\ i)$$

Remark 5 ($\mathbb{N}$ is compact).

Remark 6 ($\mathbb{N}$ is dense in $\mathbb{N}^{\mathrm{co}}$). For all $n : \mathbb{N}^{\mathrm{co}}$,

$$\mathsf{idl}\ n \text{ is an } \omega\text{-chain, } \textit{and}$$

$$\mathsf{sup}\ (\mathsf{incl} \circ \mathsf{idl}\ n) = n.$$

Remark 7 ($\mathbb{N}^{\mathrm{co}}$ is a pointed algebraic CPO with bottom element **cozero** and basis $\mathbb{N}$).

6.6.1 Coinductive Extensionality

The usual notion of propositional (Leibniz) equality in Coq is too weak to prove equalities over coinductive types such as $\mathbb{N}^{\mathrm{co}}$ (Definition 57). For example, suppose we define a function $\mathsf{coplus} : \mathbb{N}^{\mathrm{co}} \to \mathbb{N}^{\mathrm{co}} \to \mathbb{N}^{\mathrm{co}}$ for taking the sum of two conats. We quickly become stuck when trying to prove basic properties such as commutativity: $\forall n\ m,\ \mathsf{coplus}\ n\ m = \mathsf{coplus}\ m\ n$. Typically the proof would proceed by induction on either n or m, but here neither term is inductive. We cannot use coinduction either because the goal is not coinductive.

An alternative is to define a coinductive bisimulation relation that holds between n and m iff they are structurally identical:

Definition 61 ($\mathbb{N}^{\mathrm{co}}$ Equivalence)**.** *Define* $=_{\mathbb{N}^{\mathrm{co}}} : \mathbb{N}^{\mathrm{co}} \to \mathbb{N}^{\mathrm{co}} \to \mathbb{P}$ *coinductively by the inference rules:*

$$\frac{}{\mathbf{cozero} =_{\mathbb{N}^{\mathrm{co}}} \mathbf{cozero}}\ =_{\mathbb{N}^{\mathrm{co}}}\text{-ZERO} \qquad \frac{n =_{\mathbb{N}^{\mathrm{co}}} m}{\mathbf{cosucc}\ n =_{\mathbb{N}^{\mathrm{co}}} \mathbf{cosucc}\ m}\ =_{\mathbb{N}^{\mathrm{co}}}\text{-SUCC}$$

Now we can prove commutativity up to $=_{\mathbb{N}^{\mathsf{co}}}$ via coinduction: $\forall n\ m,\ \mathsf{coplus}\ n\ m =_{\mathbb{N}^{\mathsf{co}}} \mathsf{coplus}\ m\ n$. The problem with this approach, however, is that if we want to rewrite by such equations we have to explicitly prove that all of our operations on $\mathbb{N}^{\mathsf{co}}$ are proper with respect to $=_{\mathbb{N}^{\mathsf{co}}}$. But note that $=_{\mathbb{N}^{\mathsf{co}}}$ is carefully designed to coincide exactly with Leibniz equality. Although it cannot be proved within Coq, we can assert this fact in the form of an extensionality axiom (Axiom 4) that deduces Leibniz equality on conats from proofs of bisimilarity, allows us to easily rewrite by them without the need for any Proper instances [Soz09].

Axiom 4 (Conat extensionality)**.** $\forall\ n\ m : \mathbb{N}^{\mathsf{co}},\ n \simeq_{\mathbb{N}^{\mathsf{co}}} m \Rightarrow n = m.$

To gain confidence in the soundness of Axiom 4, notice that $\mathbb{N}^{\mathsf{co}}$ modulo $=_{\mathbb{N}^{\mathsf{co}}}$ is isomorphic to the type $\mathbb{N} + 1$, because every conat is either a finite natural number or the infinite conat $\omega_{\mathbb{N}}$. Let $\mathsf{sect} : \mathbb{N}^{\mathsf{co}} \to \mathbb{N} + 1$ and $\mathsf{retr} : \mathbb{N} + 1 \to \mathbb{N}^{\mathsf{co}}$ witness this isomorphism. We can derive conat extensionality as a theorem from one side of the isomorphism: $\forall n : \mathbb{N}^{\mathsf{co}},\ \mathsf{retr}\ (\mathsf{sect}\ n) = n$. That is, conat extensionality may be derived from the fact that injecting a conat into $\mathbb{N} + 1$ and then projecting it back reproduces the original conat.

Similar arguments can be made for streams (Section 6.7), cotries (Section 6.8), and cotrees (Chapter 7). See [Bou18, Section 2.2.2] and [Gro23] for more discussion on soundness of extensionality axioms for coinductive types.

6.6.2 Unlimited Fuel

When a function is not inductive on the structure of any of its arguments, a common trick (sometimes called *step-indexing* [Ahm04]) is to define it instead by induction on a separate $\mathbb{N}$ argument (the *fuel*), and ensure that enough fuel is always provided for the function to complete its task. We define fueled computations via the following ‘iter’

construction that, starting from initial element z, repeatedly applies a function f until exhausting the fuel.

Definition 62 (iter)**.** *For type A, $z : A$, $f : A \to A$, and $n : \mathbb{N}$, define* $\mathsf{iter}\ z\ f\ n : A$ *by induction on n:*

$$\begin{array}{|l} \hline \mathsf{iter}\ z\ f : \mathbb{N} \to A \\ \hline \mathbf{O} \triangleq z \\ \mathbf{S}\ n \triangleq f\ (\mathsf{iter}\ f\ n) \end{array}$$

By taking the continuous extension of a fueled iteration, we extend its domain to include $\omega_{\mathbb{N}}$, allowing it to be supplied an unlimited amount of fuel!

Definition 63 (coiter)**.** *For pointed type A and $f : A \to A$, define* $\mathsf{coiter}\ f : \mathbb{N}^{\mathsf{co}} \to A$ *by:*

$$\mathsf{coiter}\ f \triangleq (\mathsf{iter}\ \bot_A\ f)^{\mathsf{co}},$$

or when A has a top element $\top_A$, define $\hat{\mathsf{coiter}}\ f : \mathbb{N}^{\mathsf{co}} \to A$*:*

$$\hat{\mathsf{coiter}}\ f \triangleq (\mathsf{iter}\ \top_A\ f)^{\hat{\mathsf{co}}}.$$

We call this technique *lazy coiteration*, and use it to implement the Kleene closure operator (as an infinitely fueled coiteration) on a coinductive encoding of regular languages in Section 6.8.1.

The following lemma is useful for proving continuity of lazy coiterations.

Lemma 9 ((Co-)continuous coiter)**.** *Let A be an ordered type, $z : A$, and $f : A \to A$ a monotone function. Then,*

$$(\forall\ n : \mathbb{N}, z \sqsubseteq \mathsf{iter}\ z\ f\ n) \Rightarrow \mathsf{iter}\ z\ f \textit{ is monotone},$$

$$(\forall\ n : \mathbb{N}, \mathsf{iter}\ z\ f\ n \sqsubseteq z) \Rightarrow \mathsf{iter}\ z\ f \textit{ is antimonotone, and}$$

$$\mathsf{coiter}\ f \textit{ is continuous and } \hat{\mathsf{coiter}}\ f \textit{ is cocontinuous}.$$

```
coiter =
  \ o p f n ->
    case n of
      Cozero -> bot o p
      Cosucc n' -> f (coiter o p f n')
```

Figure 6.1: Haskell extraction primitive for coiter. Parameters o and p are OType and PType instance dictionary objects for the order relation of the codomain.

Coiter computation We implement an extraction primitive for coiter (shown in Figure 6.1) for lazy execution of coiterations in Haskell. Further discussion on computability and extraction of continuous extensions appears in Section 6.7.5. We justify the extraction primitive with the following generic computation lemma (from which analogous computation rules can be derived for specific coiterations):

Lemma 10 (coiter computation)**.** *Let A be a pointed CPO and $f : A \to A$ continuous. Then,*

$$\begin{aligned} \mathsf{coiter}\ f\ \mathbf{cozero} &\simeq \bot_A \\ \mathsf{coiter}\ f\ (\mathbf{cosucc}\ n) &\simeq f\ (\mathsf{coiter}\ f\ n). \end{aligned}$$

Recall that '$\simeq$' stands for order equivalence (Definition 9), which often implies propositional equality (e.g., for $\mathbb{R}^{\infty}_{\geq 0}$ by antisymmetry and $\mathbb{N}^{\mathsf{co}}$ by Axiom 4).

6.7 Streams

In this section we define *streams* (or *colists*) as a coinductive algebraic CPO (Definition 64), define a number of essential operations and predicates over them (including the oft-problematic 'filter' operation) (Section 6.7.1), and illustrate the use of the basic proof principles of AlgCo to reason about streams (Section 6.7.2), culminating in the verification of a coinductive variant of the sieve of Eratosthenes (Section 6.7.4).

Streams as an Algebraic CPO Our definition of coinductive lists deviates slightly from the standard definition of streams (e.g., [Chl22]) by inclusion of a bottom element:

Definition 64 (Streams)**.** *Define the type $\mathcal{L}_A^*$ of streams with element type A coinductively by the formation rules:*

$$\frac{}{\bot_{\mathcal{L}_A^*} : \mathcal{L}_A^*}\ \textsc{stream-bot} \qquad \frac{a : A \qquad l : \mathcal{L}_A^*}{\mathbf{cocons}\ a\ l : \mathcal{L}_A^*}\ \textsc{stream-cons}$$

We are careful not to regard $\bot_{\mathcal{L}_A^*}$ as simply a **nil** constructor for streams, viewing it instead as the *undefined* or *divergent* stream which loops forever producing no output (see Section 6.7.5 for related discussion). Streams are ordered by a straightforward structural prefix relation:

Definition 65 (Stream order)**.** *For type A, define $\sqsubseteq_{\mathcal{L}_A^*} : \mathcal{L}_A^* \to \mathcal{L}_A^* \to \mathbb{P}$ coinductively by the inference rules:*

$$\frac{l : \mathcal{L}_A^*}{\bot_{\mathcal{L}_A^*} \sqsubseteq_{\mathcal{L}_A^*} l}\ \sqsubseteq_{\mathcal{L}_A^*}\textsc{-bot} \qquad \frac{a : A \qquad l_1 \sqsubseteq_{\mathcal{L}_A^*} l_2}{\mathbf{cocons}\ a\ l_1 \sqsubseteq_{\mathcal{L}_A^*} \mathbf{cocons}\ a\ l_2}\ \sqsubseteq_{\mathcal{L}_A^*}\textsc{-cons}$$

Intuitively, we have $l_1 \sqsubseteq_{\mathcal{L}_A^*} l_2$ when either l_1 is a $\bot$-terminated finite approximation of l_2, or when l_1 and l_2 are equal. A compact basis for $\mathcal{L}_A^*$ is given by the standard inductive type $\mathcal{L}_A$ of lists (with constructors **nil** and **cons**) with prefix ordering:

Definition 66 (Lists)**.** *Define the type $\mathcal{L}_A$ of lists with element type A inductively by the formation rules:*

$$\frac{}{\mathbf{nil} : \mathcal{L}_A}\ \textsc{list-nil} \qquad \frac{a : A \qquad l : \mathcal{L}_A}{\mathbf{cons}\ a\ l : \mathcal{L}_A}\ \textsc{list-cons}$$

Definition 67 (List order)**.** *Define* $\sqsubseteq_{\mathcal{L}_A} : \mathcal{L}_A \to \mathcal{L}_A \to \mathbb{P}$ *inductively by the inference rules:*

$$\frac{}{\mathbf{nil} \sqsubseteq_{\mathcal{L}_A} l} \; \sqsubseteq_{\mathcal{L}_A}\text{-NIL} \qquad \frac{a : A \qquad l_1 \sqsubseteq_{\mathcal{L}_A} l_2}{\mathbf{cons}\, a\, l_1 \sqsubseteq_{\mathcal{L}_A} \mathbf{cons}\, a\, l_2} \; \sqsubseteq_{\mathcal{L}_A}\text{-CONS}$$

Remark 8 ($\mathcal{L}_A$ is compact)*.* For any type A, the elements of $\mathcal{L}_A$ are finite and are thus compact.

The inclusion map $\mathsf{incl}_{\mathcal{L}_A,\mathcal{L}_A^*} : \mathcal{L}_A \to \mathcal{L}_A^*$ injects lists into $\mathcal{L}_A^*$, and $\mathsf{idl}_{\mathcal{L}_A,\mathcal{L}_A^*} : \mathcal{L}_A^* \to \mathbb{N} \to \mathcal{L}_A$ generates convergent chains of list approximations of streams.

Definition 68 ($\mathsf{incl}_{\mathcal{L},\mathcal{L}^*}$)**.** *For type A and $l : \mathcal{L}_A$, define* $\mathsf{incl}_{\mathcal{L}_A,\mathcal{L}_A^*}\, l : \mathcal{L}_A^*$ *by induction on l:*

$$\begin{array}{|l} \hline \mathsf{incl}_{\mathcal{L}_A,\mathcal{L}_A^*} : \mathcal{L}_A \to \mathcal{L}_A^* \\ \hline \mathbf{nil} \triangleq \bot_{\mathcal{L}_A^*} \\ \mathbf{cons}\, a\, l \triangleq \mathbf{cocons}\, a\, (\mathsf{incl}_{\mathcal{L}_A,\mathcal{L}_A^*}\, l) \end{array}$$

Definition 69 ($\mathsf{idl}_{\mathcal{L},\mathcal{L}^*}$)**.** *For type A, $l : \mathcal{L}_A^*$, and $n : \mathbb{N}$, define* $\mathsf{idl}_{\mathcal{L}_A,\mathcal{L}_A^*}\, l\, n : \mathcal{L}_A$ *by induction on n:*

$$\begin{array}{|lll} \hline \mathsf{idl}_{\mathcal{L}_A,\mathcal{L}_A^*} : \mathcal{L}_A^* \to \mathbb{N} \to \mathcal{L}_A^* & & \\ \hline \_ & \mathbf{O} & \triangleq \mathbf{nil} \\ \bot_{\mathcal{L}_A^*} & \_ & \triangleq \mathbf{nil} \\ (\mathbf{cocons}\, a\, l) & (\mathbf{S}\, n) & \triangleq \mathbf{cons}\, a\, (\mathsf{idl}_{\mathcal{L}_A,\mathcal{L}_A^*}\, l\, n) \end{array}$$

The type $\mathcal{L}_A$ of finite lists is dense in the type $\mathcal{L}_A^*$ of streams for any type A.

Remark 9 ($\mathcal{L}_A$ is dense in $\mathcal{L}_A^*$)*.* Let A be a type. Then, for all $a : \mathcal{L}_A^*$,

$$\mathsf{idl}\; a \text{ is an } \omega\text{-chain, } \textit{and}$$

$$\mathsf{sup}\, (\mathsf{incl} \circ \mathsf{idl}\; a) = a.$$

We also have $\bot_{\mathcal{L}_A^*} \sqsubseteq l$ for all $l : \mathcal{L}_A^*$, and thus:

Remark 10 ($\mathcal{L}_A^*$ is a pointed algebraic CPO with basis $\mathcal{L}_A$).

The order relation on streams (Definition 65) is carefully chosen to allow the following extensionality axiom entailing propositional equality from order equivalence (see Section 6.6.1 for discussion of such extensionality axioms for coinductive types):

Axiom 5 (Stream extensionality)**.** *Let A be a type. Then,*

$$\forall\, l_1\ l_2 : \mathcal{L}_A^*,\ l_1 \simeq_{\mathcal{L}_A^*} l_2 \Rightarrow l_1 = l_2.$$

Streams can be easily defined by primitive corecursion (via the CoFixpoint command in Coq). For example, the following definition of nats generates a stream of natural numbers starting from an initial seed $n : \mathbb{N}$ (such that nats **O** is the stream of all natural numbers):

Example 4 (nats)**.** *For $n : \mathbb{N}$, define the stream* nats $n : \mathcal{L}_{\mathbb{N}}^*$ *of natural numbers starting from n coinductively by:*

$$\mathsf{nats}\ n \triangleq \mathbf{cocons}\ n\ (\mathsf{nats}\ (\mathbf{S}\ n)).$$

6.7.1 Cofolds

Many continuous extensions over streams share a common computational structure. In this section we define an abstraction over this common pattern ("cofolds") and use it to derive definitions and computation rules for standard operations on streams. In Section 6.7.5 we provide an extraction primitive for cofolds for lazy execution in Haskell. The foundation for cofolds is given by the standard right-associative fold operator on lists:

Definition 70 (fold)**.** *For types A and B, $z : B$, $f : A \to B \to B$, and $l : \mathcal{L}_A$, define* fold $z\ f\ l : B$ *by induction on l:*

$$\begin{array}{|l|}\hline \mathsf{fold}\ z\ f : \mathcal{L}_A \to B \\ \hline \mathbf{nil} \triangleq z \\ \mathbf{cons}\ a\ l \triangleq f\ a\ (\mathsf{fold}\ z\ f\ l) \\ \hline \end{array}$$

Operations of the form $\mathsf{fold}\ z\ f$ are often called *catamorphisms* [MFP91], or simply *folds*. We introduce *cofolds*: continuous functions on streams of the form $(\mathsf{fold}\ \bot\ f)^{\mathsf{co}}$ for monotone f, and *anticofolds* (written $\hat{\mathsf{co}}\mathsf{fold}$): cocontinuous functions of the form $(\mathsf{fold}\ \top\ f)^{\hat{\mathsf{co}}}$ for antimonotone f.

Definition 71 (cofold)**.** *For type A, pointed type B, and $f : A \to B \to B$, define* $\mathsf{cofold}\ f : \mathcal{L}_A^* \to B$ *by:*

$$\mathsf{cofold}\ f \triangleq (\mathsf{fold}\ \bot_B\ f)^{\mathsf{co}},$$

or when B has a top element $\top_B$, define $\hat{\mathsf{co}}\mathsf{fold}\ f : \mathcal{L}_A^* \to B$ *by:*

$$\hat{\mathsf{co}}\mathsf{fold}\ f \triangleq (\mathsf{fold}\ \top_B\ f)^{\hat{\mathsf{co}}}.$$

Lemma 11 ((Co)continuous cofold)**.** *Let A be a type and B an ordered type and suppose that $f\ a : B \to B$ is monotone for every $a : A$. Then,*

$$(\forall\ l : \mathcal{L}_A, z \sqsubseteq \mathsf{fold}\ z\ f\ l) \Rightarrow \mathsf{fold}\ z\ f \textit{ is monotone},$$

$$(\forall\ l : \mathcal{L}_A, \mathsf{fold}\ z\ f\ l \sqsubseteq z) \Rightarrow \mathsf{fold}\ z\ f \textit{ is antimonotone, and}$$

$$\mathsf{cofold}\ f \textit{ is continuous and } \hat{\mathsf{co}}\mathsf{fold}\ f \textit{ is cocontinuous}.$$

Although cofolds can be seen as a coinductive analogue to catamorphisms, we should be careful to distinguish them from *anamorphisms*, the true categorical dual to catamorphisms. Whereas catamorphisms provide a canonical *elimination principle for inductives*, and anamorphisms a canonical *introduction principle for coinductives*, the continuous extension construction (Lemma 5) (of which cofolds are a special case) is perhaps best understood as providing a canonical *continuous elimination principle for algebraic coinductives*.

Cofold computation The following generic computation lemma can be used to derive computation rules for cofolds.

Lemma 12 (cofold computation)**.** *Let A be a type, B an ordered type, and $f : A \to B \to B$. Then, if B is a pointed CPO and $f\ a$ is continuous for every a,*

$$\begin{aligned} \mathsf{cofold}\ f\ \bot_{\mathcal{L}^*_A} &\simeq \bot_B \\ \mathsf{cofold}\ f\ (\mathbf{cocons}\ a\ l) &\simeq f\ a\ (\mathsf{cofold}\ f\ l), \end{aligned}$$

or if B is an lCPO with a top element and $f\ a$ is l-continuous for every a,

$$\begin{aligned} \hat{\mathsf{cofold}}\ f\ \bot_{\mathcal{L}^*_A} &\simeq \top_B \\ \hat{\mathsf{cofold}}\ f\ (\mathbf{cocons}\ a\ l) &\simeq f\ a\ (\hat{\mathsf{cofold}}\ f\ l). \end{aligned}$$

Example cofolds We present some illustrative cofolds over streams and derive their computation rules from Lemma 12. We sometimes give explicit names to the basis functions being continuously extended to make proofs about them more readable (e.g., Definition 81 and Lemma 13). Our first example is the cofold $\mathsf{length}_{\mathcal{L}^*_\mathsf{A}}$ mapping streams to $\mathbb{N}^{\mathsf{co}}$ such that the length of any infinite stream is equal to $\omega_{\mathbb{N}}$.

Definition 72 ($\mathsf{length}_{\mathcal{L}^*}$)**.** *For type A, define*

$\mathsf{length}_{\mathcal{L}^*_\mathsf{A}} : \mathcal{L}^*_A \to \mathbb{N}^{\mathsf{co}} \triangleq (\mathsf{fold}\ \mathbf{cozero}\ (\lambda\_.\ \mathbf{cosucc}))^{\mathsf{co}}$*, with computation rules:*

$$\begin{aligned} \mathsf{length}_{\mathcal{L}^*_\mathsf{A}}\ \bot_{\mathcal{L}^*_A} &= \mathbf{cozero} \\ \mathsf{length}_{\mathcal{L}^*_\mathsf{A}}\ (\mathbf{cocons}\ \_\ l) &= \mathbf{cosucc}\ (\mathsf{length}_{\mathcal{L}^*_\mathsf{A}}\ l). \end{aligned}$$

And mapping a function over a stream:

Definition 73 ($\mathsf{map}_{\mathcal{L}^*}$)**.** *For types A and B, and $f : A \to B$, define*

$\mathsf{map}_{\mathcal{L}^*_\mathsf{A}}\ f : \mathcal{L}^*_A \to \mathcal{L}^*_B \triangleq (\mathsf{map}_{\mathcal{L}_\mathsf{A}}\ f)^{\mathsf{co}}$*, where:*

$$\mathsf{map}_{\mathcal{L}_\mathsf{A}}\ f \triangleq \mathsf{fold}\ \bot_{\mathcal{L}^*_B}\ (\lambda a.\ \lambda l.\ \mathbf{cocons}\ (f\ a)\ l)$$

with computation rules:

$$\begin{aligned}
\mathsf{map}_{\mathcal{L}^*_{\mathsf{A}}}\ f\ \bot_{\mathcal{L}^*_A} &= \bot_{\mathcal{L}^*_A} \\
\mathsf{map}_{\mathcal{L}^*_{\mathsf{A}}}\ f\ (\mathbf{cocons}\ a\ l) &= \mathbf{cocons}\ (f\ a)\ (\mathsf{map}_{\mathcal{L}^*_{\mathsf{A}}}\ f\ l).
\end{aligned}$$

The infinite sum over a stream of extended reals is expressed as the cofold $\mathsf{sum}_{\mathcal{L}^*_{\mathbb{R}^\infty_{\geq 0}}}$ mapping streams of extended reals into $\mathbb{R}^\infty_{\geq 0}$:

Definition 74 ($\mathsf{sum}_{\mathcal{L}^*_{\mathbb{R}^\infty_{\geq 0}}}$)**.** *Define* $\mathsf{sum}_{\mathcal{L}^*_{\mathbb{R}^\infty_{\geq 0}}} : \mathcal{L}^*_{\mathbb{R}^\infty_{\geq 0}} \to \mathbb{R}^\infty_{\geq 0} \triangleq \mathsf{sum}_{\mathcal{L}_{\mathbb{R}^\infty_{\geq 0}}}{}^{\mathsf{co}}$, *where:*

$$\mathsf{sum}_{\mathcal{L}_{\mathbb{R}^\infty_{\geq 0}}} \triangleq \mathsf{fold}\ 0\ (\lambda x.\ \lambda y.\ x + y)$$

with computation rules:

$$\begin{aligned}
\mathsf{sum}_{\mathcal{L}^*_{\mathbb{R}^\infty_{\geq 0}}}\ \bot_{\mathcal{L}^*_A} &= 0 \\
\mathsf{sum}_{\mathcal{L}^*_{\mathbb{R}^\infty_{\geq 0}}}\ (\mathbf{cocons}\ a\ l) &= a + \mathsf{sum}_{\mathcal{L}^*_{\mathbb{R}^\infty_{\geq 0}}}\ l.
\end{aligned}$$

Quantifying Predicates Over Streams A common task is to assert that a property holds for some element appearing in a stream. We define existential quantification over streams as a continuous property as follows:

Definition 75 ($\exists^{\mathsf{co}}_P$)**.** *For type* A *and predicate* $P : A \to \mathbb{P}$, *define* $\exists^{\mathsf{co}}_P : \mathcal{L}^*_A \to \mathbb{P} \triangleq (\mathsf{fold}\ \bot\ (\lambda a : A.\ \lambda Q : \mathbb{P}.\ P\ a \vee Q))^{\mathsf{co}}$, *with derived introduction and elimination rules:*

$\exists^{\mathsf{co}}_P$-INTRO-1

$$\frac{P\ a \qquad l : \mathcal{L}^*_A}{\exists^{\mathsf{co}}_P\ (\mathbf{cocons}\ a\ l)}$$

$\exists^{\mathsf{co}}_P$-INTRO-2

$$\frac{a : A \qquad \exists^{\mathsf{co}}_P\ l}{\exists^{\mathsf{co}}_P\ (\mathbf{cocons}\ a\ l)}$$

$\exists^{\mathsf{co}}_P$-ELIM

$$\frac{\exists^{\mathsf{co}}_P\ (\mathbf{cocons}\ a\ l)}{P\ a \vee \exists^{\mathsf{co}}_P\ l}$$

Dually, we define universal quantification over streams as a cocontinuous property as follows:

Definition 76 ($\forall_P^{\hat{\mathfrak{co}}}$)**.** *For type A and predicate $P : A \to \mathbb{P}$, define* $\forall_P^{\hat{\mathfrak{co}}} : \mathcal{L}_A^* \to \mathbb{P} \triangleq (\mathsf{fold}\ \top\ (\lambda a : A.\ \lambda Q : \mathbb{P}.\ P\ a \wedge Q))^{\hat{\mathfrak{co}}}$*, with derived introduction and elimination rules:*

$\forall_P^{\hat{\mathfrak{co}}}$-INTRO

$$\frac{P\ a \qquad \forall_P^{\hat{\mathfrak{co}}}\ l}{\forall_P^{\hat{\mathfrak{co}}}\ (\mathbf{cocons}\ a\ l)}$$

$\forall_P^{\hat{\mathfrak{co}}}$-ELIM-1

$$\frac{\forall_P^{\hat{\mathfrak{co}}}\ (\mathbf{cocons}\ a\ l)}{P\ a}$$

$\forall_P^{\hat{\mathfrak{co}}}$-ELIM-2

$$\frac{\forall_P^{\hat{\mathfrak{co}}}\ (\mathbf{cocons}\ a\ l)}{\forall_P^{\hat{\mathfrak{co}}}\ l}$$

Definitions 75 and 76 are used to prove the correctness properties of the sieve of Eratosthenes in Theorems 16 and 17.

Order Relation as Continuous Extension Although a primitive order relation on streams must have already been defined (Definition 65) to gain access to the machinery of the AlgCo framework in the first place, we can re-define it as a continuous extension and prove it equivalent to the original.

Definition 77 ($\sqsubseteq_{\mathcal{L}}^{\hat{\mathfrak{co}}}$)**.** *For type A, define* $\sqsubseteq_{\mathcal{L}_A}^{\hat{\mathfrak{co}}} : \mathcal{L}_A^* \to \mathcal{L}_A^* \to \mathbb{P}$ *where* $\sqsubseteq_{\mathcal{L}_A} : \mathcal{L}_A \to \mathcal{L}_A^* \triangleq$

$$\begin{aligned}
&\mathsf{fold}\ (\lambda_.\ \top)\ (\lambda a.\ \lambda f.\ \lambda l.\ \mathit{match}\ l\ \mathit{with}\\
&\qquad \mid \bot_{\mathcal{L}_A^*} \Rightarrow \bot\\
&\qquad \mid \mathbf{cocons}\ b\ l' \Rightarrow a = b \wedge f\ l'\\
&\qquad \mathit{end})
\end{aligned}$$

Remark 11 ($\sqsubseteq_{\mathcal{L}}^{\hat{\mathfrak{co}}}$ coincides with $\sqsubseteq_{\mathcal{L}^*}$)*.* Let A be a type. Then,

$$\forall\ l_1\ l_2 : \mathcal{L}_A^*,\ l_1 \sqsubseteq_{\mathcal{L}_A}^{\hat{\mathfrak{co}}} l_2 \iff l_1 \sqsubseteq_{\mathcal{L}_A^*} l_2.$$

Definition 77 is often more convenient in practice than the primitive order relation (Definition 65) because it can be fused (via Theorem 14) with other continuous extensions. Reasoning with the primitive order typically requires proof by coinduction via the primitive 'cofix' tactic which suffers from the problems described in Section 6.1 (e.g.,

rigid coinduction hypotheses with overly constraining syntactic rules for successful application). Definition 77, on the other hand, can be reasoned about inductively as with any other continuous extension.

The following definition of the cocontinuous property $\mathsf{ordered}_{\mathcal{L}^*}$ states that a stream is well-ordered with respect to a given relation R. We then specialize $\mathsf{ordered}_{\mathcal{L}^*}$ to particular choices of R to implement sorted and nodup predicates.

Definition 78 ($\mathsf{ordered}_{\mathcal{L}^*}$)**.** *For type A and relation $R : A \to A \to \mathbb{P}$, define $\mathsf{ordered}_{\mathcal{L}^*_{\mathsf{A}}}\ R : \mathcal{L}^*_A \to \mathbb{P} \triangleq (\mathsf{ordered}_{\mathcal{L}_{\mathsf{A}}}\ R)^{\mathsf{co}}$, where $\mathsf{ordered}_{\mathcal{L}_{\mathsf{A}}}\ R : \mathcal{L}_A \to \mathbb{P}$ is given inductively by the inference rules:*

$$\frac{}{\mathsf{ordered}_{\mathcal{L}_{\mathsf{A}}}\ R\ \mathbf{nil}}\ \textsc{ordered-nil} \qquad \frac{\forall_{(R\,a)}\ l \qquad \mathsf{ordered}_{\mathcal{L}_{\mathsf{A}}}\ l}{\mathsf{ordered}_{\mathcal{L}_{\mathsf{A}}}\ (\mathbf{cocons}\ a\ l)}\ \textsc{ordered-cons}$$

Definition 79 ($\mathsf{sorted}_{\mathcal{L}^*}$)**.** *For ordered type A, Define*

$$\mathsf{sorted}_{\mathcal{L}^*_{\mathsf{A}}} : \mathcal{L}^*_A \to \mathbb{P} \triangleq \mathsf{ordered}_{\mathcal{L}^*_{\mathsf{A}}}\ (\leq_A).$$

Definition 80 ($\mathsf{nodup}_{\mathcal{L}^*}$)**.** *For ordered type A, Define*

$$\mathsf{nodup}_{\mathcal{L}^*_{\mathsf{A}}} : \mathcal{L}^*_A \to \mathbb{P} \triangleq \mathsf{ordered}_{\mathcal{L}^*_{\mathsf{A}}}\ (\lambda x.\ \lambda y.\ x \neq y).$$

Note that the cofold construction as presented in this dissertation (Definition 103) is not sufficient for implementing $\mathsf{ordered}_{\mathcal{L}^*_{\mathsf{A}}}$. This is not a fundamental limitation; cofolds can be generalized to an analogue of "paramorphisms" [MFP91] which suffice to define properties like $\mathsf{ordered}_{\mathcal{L}^*_{\mathsf{A}}}$. We favor the simpler formulation of cofold given in Definition 103 for clarity of presentation.

Cofilter Filtering a coinductive stream by a given predicate is a notoriously awkward exercise, typically requiring the stream type to be extended with a special constructor for so-called "silent-steps" (as in, e.g., [XZH+20], inducing significant definitional clutter and

performance overhead), or the use of complicated mixed well-founded induction-coinduction schemes as in [Ber05]. We easily define filter as a cofold:

Definition 81 ($\mathsf{filter}_{\mathcal{L}^*}$)**.** *For type A and $f : A \to \mathbb{B}$, define* $\mathsf{filter}_{\mathcal{L}^*_\mathsf{A}}\ f : \mathcal{L}^*_A \to \mathcal{L}^*_A \triangleq (\mathsf{filter}_{\mathcal{L}_\mathsf{A}}\ f)^{\mathsf{co}}$*, where:*

$$\mathsf{filter}_{\mathcal{L}_\mathsf{A}}\ f : \mathcal{L}_A \to \mathcal{L}^*_A \triangleq \mathsf{fold}\ \bot\ (\lambda a.\ \lambda l.\ \mathit{if}\ f\ a\ \mathit{then}\ \mathbf{cocons}\ a\ l\ \mathit{else}\ l)$$

with computation rules:

$$\begin{aligned} \mathsf{filter}_{\mathcal{L}^*_\mathsf{A}}\ f\ \bot_{\mathcal{L}^*_A} &= \bot_{\mathcal{L}^*_A} \\ \mathsf{filter}_{\mathcal{L}^*_\mathsf{A}}\ f\ (\mathbf{cocons}\ a\ l) &= \mathit{if}\ f\ a\ \mathit{then}\ \mathbf{cocons}\ a\ (\mathsf{filter}_{\mathcal{L}^*_\mathsf{A}}\ f\ l)\ \mathit{else}\ \mathsf{filter}_{\mathcal{L}^*_\mathsf{A}}\ f\ l. \end{aligned}$$

6.7.2 Proving With Fusion

Here we demonstrate a common technique for proving equations between continuous functions over algebraic CPOs by proving commutativity of $\mathsf{filter}_{\mathcal{L}^*_\mathsf{A}}$. First we use Theorem 14 to fuse both sides of the equation, and then we apply Lemma 6 to reduce the proof to induction over basis elements.

Lemma 13 ($\mathsf{filter}_{\mathcal{L}^*}$ is commutative)**.** *Let A be a type and $f, g : A \to \mathbb{B}$. Then,*

$$\forall\ l : \mathcal{L}^*_A, \mathsf{filter}_{\mathcal{L}^*_\mathsf{A}}\ f\ (\mathsf{filter}_{\mathcal{L}^*_\mathsf{A}}\ g\ l) = \mathsf{filter}_{\mathcal{L}^*_\mathsf{A}}\ g\ (\mathsf{filter}_{\mathcal{L}^*_\mathsf{A}}\ f\ l).$$

Proof. Unfolding the definition of $\mathsf{filter}_{\mathcal{L}^*_\mathsf{A}}$*, we wish to show:*

$$\mathsf{filter}_{\mathcal{L}^*_\mathsf{A}}\ f\ \circ (\mathsf{filter}_{\mathcal{L}_\mathsf{A}}\ g)^{\mathsf{co}} = \mathsf{filter}_{\mathcal{L}^*_\mathsf{A}}\ g\ \circ (\mathsf{filter}_{\mathcal{L}_\mathsf{A}}\ f)^{\mathsf{co}}.$$

We first fuse both sides of the equation by Theorem 14:

$$(\mathsf{filter}_{\mathcal{L}^*_\mathsf{A}}\ f \circ \mathsf{filter}_{\mathcal{L}_\mathsf{A}}\ g)^{\mathsf{co}} = (\mathsf{filter}_{\mathcal{L}^*_\mathsf{A}}\ g \circ \mathsf{filter}_{\mathcal{L}_\mathsf{A}}\ f)^{\mathsf{co}}$$

which then follows by Corollary 6 from:

$$\forall\ l : \mathcal{L}_A, \mathsf{filter}_{\mathcal{L}^*_\mathsf{A}}\ f\ (\mathsf{filter}_{\mathcal{L}_\mathsf{A}}\ g\ l) = \mathsf{filter}_{\mathcal{L}^*_\mathsf{A}}\ g\ (\mathsf{filter}_{\mathcal{L}_\mathsf{A}}\ f\ l)$$

which follows by straightforward induction on l and application of the $\mathsf{filter}_{\mathcal{L}^*_\mathsf{A}}$ *computation rule (Definition 81).* □

6.7.3 Proving (Co-)Continuous Properties

A surprising but elegant feature of the AlgCo framework is that by treating predicates (i.e., *propositional functions*) over algebraic CPOs as a special case of continuous functions, we can easily compose them with continuous extensions and greatly simplify their proofs via fusion. To illustrate, let us prove that $\mathsf{filter}_{\mathcal{L}^*}\ P$ preserves $\mathsf{ordered}_{\mathcal{L}^*}\ R$ for any P and R:

Lemma 14 ($\mathsf{filter}_{\mathcal{L}^*}$ preserves $\mathsf{ordered}_{\mathcal{L}^*}$)**.** *Let A be a type, $R : A \to A \to \mathbb{P}$, $P : A \to \mathbb{P}$, and $s : \mathcal{L}^*_A$ such that $\mathsf{ordered}_{\mathcal{L}^*_A}\ R\ s$. Then,*

$$\mathsf{ordered}_{\mathcal{L}^*_A}\ R\ (\mathsf{filter}_{\mathcal{L}^*_A}\ P\ s).$$

Proof. Unfolding the definition of $\mathsf{filter}_{\mathcal{L}^*_A}$, the goal becomes:

$$\mathsf{ordered}_{\mathcal{L}^*_A}\ R\ ((\mathsf{filter}_{\mathcal{L}_A}\ P)^{\mathsf{co}}\ s).$$

By fusion (Theorem 15), this is equivalent to:

$$(\mathsf{ordered}_{\mathcal{L}^*_A}\ R \circ \mathsf{filter}_{\mathcal{L}_A}\ P)^{\hat{\mathsf{co}}}\ s$$

which by $\hat{\mathsf{co}}$-intro (Definition 56) follows from:

$$\forall i : \mathbb{N},\ \mathsf{ordered}_{\mathcal{L}^*_A}\ R\ (\mathsf{filter}_{\mathcal{L}_A}\ P\ (\mathsf{idl}\ s\ i)).$$

Fix i. By the fact that $\forall l : \mathcal{L}_A, \mathsf{ordered}_{\mathcal{L}_A}\ l \Rightarrow \mathsf{ordered}_{\mathcal{L}^*_A}\ (\mathsf{filter}_{\mathcal{L}_A}\ P\ l)$ (by straightforward induction on l), it suffices to show:

$$\mathsf{ordered}_{\mathcal{L}_A}\ R\ (\mathsf{idl}\ s\ i)$$

which follows by $\hat{\mathsf{co}}$-elim (Definition 56) from $\mathsf{ordered}_{\mathcal{L}^*_A}\ R\ s$. □

6.7.4 Sieve of Eratosthenes

The sieve of Eratosthenes is an ancient algorithm for generating sequences of prime numbers up to a given limit. An infinitary variant of the sieve (sometimes called "the unfaithful sieve" [O'N09]) is often used to demonstrate the elegance of lazy functional programming. However, it is difficult to replicate in a logically sound type theory (such as Coq's) due to its use of a filter operation on streams. We define the sieve as the continuous extension of a monotone function on lists:

Definition 82 (sieve)**.** *Define* sieve : $\mathcal{L}^*_{\mathbb{Z}} \triangleq$ sieve_aux (nats 2)*, where:*

$$\text{sieve\_aux} : \mathcal{L}^*_{\mathbb{Z}} \to \mathcal{L}^*_{\mathbb{Z}} \triangleq (\text{sieve\_aux}_{\mathcal{L}_{\mathsf{A}}})^{\mathsf{co}}\text{, where}$$

$$\text{sieve\_aux}_{\mathcal{L}_{\mathsf{A}}} \triangleq \text{fold} \perp_{\mathcal{L}^*_{\mathbb{Z}}} (\lambda n.\ \lambda l.\ \textbf{cocons}\ n\ (\text{filter}_{\mathcal{L}^*_{\mathsf{A}}}\ (\lambda m.\ m \bmod n \neq 0)\ l))$$

with computation rule:

$$\text{sieve\_aux}\ (\textbf{cocons}\ n\ l) = \textbf{cocons}\ n\ (\text{filter}_{\mathcal{L}^*_{\mathsf{A}}}\ (\lambda m.\ m \bmod n \neq 0)\ (\text{sieve\_aux}\ l)).$$

We verify sieve by proving that it is complete (Theorem 16) and sound (Theorem 17) with respect to the prime numbers, and that the numbers produced by sieve appear in ascending order with no duplicates (Theorem 18). We work with the following definition of primality:

Definition 83 (is_prime)**.** *Define* is_prime : $\mathbb{Z} \to \mathbb{P}$ *by:*

$$\text{is\_prime} \triangleq 1 < n \wedge \forall\, m,\ 1 < m \to n \neq m \to n \bmod m \neq 0.$$

Theorem 16 (sieve is complete with respect to the prime numbers)**.** *Let* $n : \mathbb{Z}$ *such that* is_prime *n. Then,* $\exists^{\mathsf{co}}_{\mathsf{eq}\,n}$ sieve*. I.e., every prime number n appears in the stream generated by* sieve*.*

Theorem 16 states a continuous property (Definition 55) of the sieve stream, and so is proved via the introduction rule co-intro for continuous properties by showing that it holds for every n for *some* finite approximation of the sieve.

Theorem 17 (sieve is sound with respect to the prime numbers)**.** $\forall^{\hat{\mathsf{co}}}_{\mathsf{is\_prime}}\ \mathsf{sieve}$. *I.e., every number appearing in the stream generated by* sieve *is prime.*

Theorem 17 states a cocontinuous property (Definition 56) of the sieve stream, and so is proved via the introduction rule ĉo-intro for cocontinuous properties by showing that it holds for *all* finite approximations of the sieve.

Theorem 18 (sieve is sorted and contains no duplicates)**.** $\mathsf{sorted}_{\mathcal{L}^*_{\mathbb{Z}}}\ \mathsf{sieve}$ *and* $\mathsf{nodup}_{\mathcal{L}^*_{\mathbb{Z}}}\ \mathsf{sieve}$.

Theorem 18 is proved by showing that the stream nats 2 is strictly increasing and that the continuous extension sieve_aux (Definition 82) is strictly order preserving.

6.7.5 Extracting the Sieve

Although continuous extensions are not computable in general due to their non-constructive definition (via the sup operator in Section 3.2), we implement an extraction primitive for cofolds over streams (shown in Figure 6.2) for lazy execution in Haskell. The correctness of the cofold extraction primitive is justified by the cofold computation rule in Lemma 12 and can be checked explicitly on streams intended for execution (see, e.g., the sieve computation rule in Definition 82).

Computability of Continuous Extensions The extraction primitive shown in Figure 6.2 provides a computational interpretation of cofolds that is only partially correct because programs extracted from continuous extensions are not guaranteed to terminate for every input (cf. the extracted fixpoint operators of [Cha10] and [BK08]). The reason is that continuous predicates are only *semi-decidable* (see Section 6.5), and continuity in

```
cofold =
  \ o p f l ->
    case l of
      Conil -> bot o p
      Cocons a l' -> f a (cofold o p f l')
```

Figure 6.2: Haskell extraction primitive for cofold. Parameters o and p are OType and PType instance dictionary objects for the order relation of the codomain.

general corresponds only to *partial computability*. For example, the continuous Boolean-valued predicate bad $\triangleq (\lambda\_\ x.\ x)^{\mathsf{co}} : \mathcal{L}^*_{\mathbb{B}} \to \mathbb{B}$ diverges for every stream.

We may, however, define a notion of *productivity* of streams generated by continuous extensions (the proof of which for our sieve follows from Euclid's theorem on the infinitude of primes [H+56]):

Definition 84 (productive)**.** *For any type A, a stream* $s : \mathcal{L}^*_A$ *is said to be (infinitely) productive when* $\mathsf{length}_{\mathcal{L}^*_A}\ s = \omega_{\mathbb{N}}$.

We prove that sieve is productive:

Theorem 19 (sieve is productive)**.** $\mathsf{length}_{\mathcal{L}^*_{\mathbb{Z}}}\ \mathsf{sieve} = \omega_{\mathbb{N}}$.

But this notion of productivity is not sufficient in general to absolutely guarantee the absence of divergence. To see why, consider the stream defined coinductively by bad_stream $\triangleq$ **cocons** (bad (nats 0)) bad_stream. Each element of bad_stream is provably equal to **false**, and bad_stream is productive, but any attempt to extract and compute with its elements will immediately enter an infinite loop.

6.8 Coinductive Tries

Formal languages, typically defined as sets of strings over an alphabet Σ, can be given an elegant representation by infinite prefix trees (*cotries*) branching over Σ. The

cotrie representation of a formal language encodes an automaton for recognizing strings in the language by marking each node as either accepting or rejecting the empty string ϵ and directly encoding the Brzozowski derivative at the node with respect to each symbol of the alphabet via a coinductive continuation [Tra15]. In this section, we define the type of cotries as an algebraic CPO, develop regular operations over them, and prove that the operations satisfy the axioms of Kleene algebra, resulting in extraction of a verified regular expression library to Haskell.

Formal languages as an algebraic CPO The type of cotries is defined with respect to a finite alphabet type (indexing the children of each node) and a pointed ordered type for node labels. For ease of presentation, we specialize to a fixed finite alphabet Σ and label type $\mathbb{B}$.

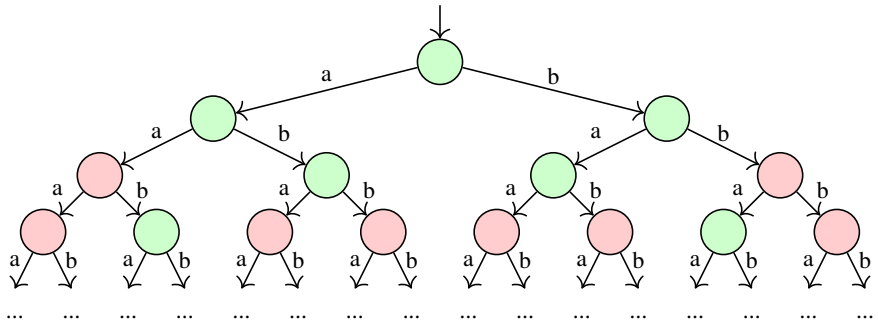


Figure 6.3: Coinductive trie encoding of regular language '$\epsilon + a^*b + b^*a$' over alphabet {a, b}. Green nodes indicate accept states of the encoded automaton.

Definition 85 (lang_Σ)**.** *Define the type* lang_Σ *of formal languages over finite alphabet* Σ *coinductively by the formation rule:*

$$\frac{b : \mathbb{B} \qquad \forall\, a : \Sigma,\ k\ a : \mathsf{lang}_\Sigma}{\textbf{lnode}\ b\ k : \mathsf{lang}_\Sigma}\ \textsc{lang-node}$$

Definition 86 (lang_Σ order)**.** *Define* $\sqsubseteq_{\mathsf{lang}_\Sigma} : \mathsf{lang}_\Sigma \to \mathsf{lang}_\Sigma \to \mathbb{P}$ *coinductively by the inference rule:*

$$\frac{b_1 \sqsubseteq_{\mathbb{B}} b_2 \qquad \forall\, a : \Sigma,\ f\ a \sqsubseteq_{\mathsf{lang}_\Sigma} g\ a}{\mathbf{lnode}\ b_1\ f \sqsubseteq_{\mathsf{lang}_\Sigma} \mathbf{lnode}\ b_2\ g}\ \sqsubseteq_{\mathsf{lang}_\Sigma}\text{-NODE}$$

Unlike the other coinductive types in this dissertation, there is no constructor for $\bot$, or any finite constructor at all (provided that the alphabet is nonempty). Every cotrie is thus infinite and congruent in structure to every other cotrie, differing only by the values of node labels. The empty language, encoded by a cotree in which every label is false, naturally serves as the bottom element:

Definition 87 ($\emptyset$ language)**.** *Define the empty language* $\emptyset$ *coinductively by:*

$$\emptyset \triangleq \mathbf{lnode}\ \mathbf{false}\ (\lambda\_.\ \emptyset).$$

Although all cotries are infinite, we can take as a compact basis the subset of cotries that are meaningfully defined only up to a finite depth (being equal to $\emptyset$ thereafter). Such cotries are finitely approximable and thus compact, and can be represented by an inductive type with a $\bot$ constructor standing in for $\emptyset$:

Definition 88 (tlang_Σ)**.** *Define the type* tlang_Σ *of finite approximations of trie languages over alphabet* Σ *inductively by the formation rules:*

$$\frac{}{\bot_{\mathsf{tlang}_\Sigma} : \mathsf{tlang}_\Sigma}\ \text{TLANG-BOT} \qquad \frac{b : \mathbb{B} \qquad \forall\, a : \Sigma,\ k\ a : \mathsf{tlang}_\Sigma}{\mathbf{tlnode}\ k : \mathsf{tlang}_\Sigma}\ \text{TRIE-NODE}$$

However, if we were to define the order relation on tlang_Σ in the usual way (e.g., Definition 65), the inclusion map into lang_Σ would not be injective (up to order equivalence), as multiple elements of tlang_Σ (all approximations of $\emptyset$) would be mapped to

$\emptyset$. We solve this by defining the order relation in such a way that all approximations of the bottom cotrie are collapsed into a single equivalence class.

Definition 89 (tlang_Σ order)**.** *Define* $\sqsubseteq_{\mathsf{tlang}_\Sigma} : \mathsf{tlang}_\Sigma \to \mathsf{tlang}_\Sigma \to \mathbb{P}$ *inductively by the inference rules:*

$$\frac{\mathsf{is\_bot}\ t_1 \qquad t_2 : \mathsf{tlang}_\Sigma}{t_1 \sqsubseteq_{\mathsf{tlang}_\Sigma} t_2}\ \sqsubseteq_{\mathsf{tlang}_\Sigma}\text{-BOT} \qquad \frac{b_1 \sqsubseteq b_2 \qquad \forall\, a : \Sigma,\ f\ a \sqsubseteq_{\mathsf{tlang}_\Sigma} g\ a}{\mathbf{tlnode}\ b_1\ f \sqsubseteq_{\mathsf{tlang}_\Sigma} \mathbf{tlnode}\ b_2\ g}\ \sqsubseteq_{\mathsf{tlang}_\Sigma}\text{-NODE}$$

where $\mathsf{is\_bot} : \mathsf{tlang}_\Sigma \to \mathbb{P}$ *is defined inductively by:*

$$\frac{}{\mathsf{is\_bot}\ \bot_{\mathsf{tlang}_\Sigma}}\ \mathsf{is\_bot}\text{-BOT} \qquad \frac{\forall\, a : \Sigma, \mathsf{is\_bot}\ (k\ a)}{\mathsf{is\_bot}\ (\mathbf{tlnode}\ \mathbf{false}\ k)}\ \mathsf{is\_bot}\text{-NODE}$$

We prove that tlang_Σ is compact with respect to the above order relation.

Remark 12 (tlang_Σ is compact).

The inclusion map $\mathsf{incl} : \mathsf{tlang}_\Sigma \to \mathsf{lang}_\Sigma$ injects tlang_Σ into lang_Σ, and $\mathsf{idl} : \mathsf{lang}_\Sigma \to \mathbb{N} \to \mathsf{tlang}_\Sigma$ generates convergent chains of finite approximations of language cotries.

Definition 90 ($\mathsf{incl}_{\mathsf{tlang}_\Sigma,\mathsf{lang}_\Sigma}$)**.** *For type A and* $l : \mathsf{tlang}_\Sigma$*, define* $\mathsf{incl}_{\mathsf{tlang}_\Sigma,\mathsf{lang}_\Sigma}\ l : \mathsf{lang}_\Sigma$ *by induction on l:*

$$\begin{array}{l} \mathsf{incl}_{\mathsf{tlang}_\Sigma,\mathsf{lang}_\Sigma} : \mathsf{tlang}_\Sigma \to \mathsf{lang}_\Sigma \\ \hline \bot_{\mathsf{tlang}_\Sigma} \triangleq \emptyset \\ \mathbf{tlnode}\ b\ k \triangleq \mathbf{lnode}\ b\ (\mathsf{incl}_{\mathsf{tlang}_\Sigma,\mathsf{lang}_\Sigma} \circ k) \end{array}$$

Definition 91 ($\mathsf{idl}_{\mathsf{tlang}_\Sigma,\mathsf{lang}_\Sigma}$)**.** *For type* A, $l : \mathsf{lang}_\Sigma$, *and* $n : \mathbb{N}$, *define* $\mathsf{idl}_{\mathsf{tlang}_\Sigma,\mathsf{lang}_\Sigma}\ l\ n : \mathsf{tlang}_\Sigma$ *by induction on* n*:*

$$\mathsf{idl}_{\mathsf{tlang}_\Sigma,\mathsf{lang}_\Sigma} : \mathcal{L}_A^* \to \mathbb{N} \to \mathcal{L}_A^*$$

$$\_\ \ \mathbf{O} \triangleq \bot_{\mathsf{tlang}_\Sigma}$$

$$(\mathbf{lnode}\ b\ k)\ \ (\mathbf{S}\ n) \triangleq \mathbf{tlnode}\ b\ (\lambda x.\ \mathsf{idl}_{\mathsf{tlang}_\Sigma,\mathsf{lang}_\Sigma}\ (k\ x)\ n)$$

Remark 13 (tlang_Σ is dense in tlang_Σ). For all $l : \mathsf{lang}_\Sigma$,

$$\mathsf{idl}\ l \text{ is an } \omega\text{-chain, } \textit{and}$$

$$\mathsf{sup}\ (\mathsf{incl} \circ \mathsf{idl}\ l) = l.$$

We also have $\emptyset \sqsubseteq l$ for all $l : \mathsf{lang}_\Sigma$, and thus:

Remark 14 (lang_Σ is a pointed algebraic CPO with basis tlang_Σ).

As with conats (Section 6.6) and streams (Section 6.7), the order relation on cotries (Definition 86) is carefully chosen to allow the safe addition of an extensionality axiom entailing propositional equality from order equivalence (see Section 6.6.1 for discussion of such extensionality axioms for coinductive types):

Axiom 6 (lang_Σ extensionality)**.** $\forall\ l_1\ l_2 : \mathsf{lang}_\Sigma,\ l_1 \simeq_{\mathsf{lang}_\Sigma} l_2 \Rightarrow l_1 = l_2$.

6.8.1 Regular Languages

The cotrie encoding of formal languages naturally leads to a straightforward algorithm for checking membership of a string $s : \mathcal{L}_\Sigma$ by induction on s.

Definition 92 (lang_Σ membership)**.** *For* $t : \mathsf{lang}_\Sigma$ *and* $l : \mathcal{L}_\Sigma$, *define* $t \in l : \mathbb{B}$ *by induction on* l*:*

$$\in : \mathsf{lang}_\Sigma \to \mathcal{L}_\Sigma \to \mathbb{B}$$

$$(\mathbf{lnode}\ b\ \_)\ \ \mathbf{nil} \triangleq b$$

$$(\mathbf{lnode}\ \_\ k)\ \ (\mathbf{cons}\ a\ l) \triangleq \mathsf{in\_lang}\ (k\ a)\ l$$

A remarkable property of language cotries is that the semantic notion of language equivalence coincides precisely with structural equality (cf. the extensional tries of [AL21]). This greatly simplifies proofs of equalities between cotries by reducing them to straightforward induction over lists.

Lemma 15 (Semantic equivalence coincides with structural equality)**.** *Let* $a : \mathsf{lang}_\Sigma$ *and* $b : \mathsf{lang}_\Sigma$. Σ. *Then,*

$$(\forall l : \mathcal{L}_\Sigma,\ a \in l = b \in l) \iff a = b.$$

We let '$\mathsf{o}\ a$' denote the Boolean indicating whether a accepts the empty string or not and we let '$\delta_a\ x$' denote the Brzozowski derivative of language a with respect to character x (i.e., $\mathsf{o}\ (\mathbf{lnode}\ b\ \_) \triangleq b$ for all b and $\delta_{(\mathbf{lnode}\ \_\ \mathbf{k})}\ x \triangleq k\ x$ for all k and x).

Definition 93 (Regular language constructions)**.** *Define the language* ϵ *containing only the empty string by:*

$$\epsilon \triangleq \mathbf{lnode}\ \mathbf{true}\ (\lambda\_.\ \emptyset),$$

define the union and intersection of $a : \mathsf{lang}_\Sigma$ *and* $b : \mathsf{lang}_\Sigma$ *by primitive coinduction:*

$$a + b \triangleq \mathbf{lnode}\ (\mathsf{o}\ a \vee \mathsf{o}\ b)\ (\lambda x.\ \delta_a\ x + \delta_b\ x)$$
$$a\ \&\ b \triangleq \mathbf{lnode}\ (\mathsf{o}\ a \wedge \mathsf{o}\ b)\ (\lambda x.\ \delta_a\ x\ \&\ \delta_b\ x),$$

and define the complement of $a : \mathsf{lang}_\Sigma$ *by primitive coinduction:*

$$\neg a \triangleq \mathbf{lnode}\ (\neg(\mathsf{o}\ a))\ (\lambda x.\ \neg(\delta_a\ x)).$$

The structural order on cotrie languages also coincides with the standard semantic order on Kleene algebras: $a \sqsubseteq b \iff a + b = a$.

The concatenation and Kleene star operators are more difficult to implement than those given above because they are not primitive corecursive. We can attempt to define the concatenation of $a : \mathsf{lang}_\Sigma$ and $b : \mathsf{lang}_\Sigma$ as follows:

$$a \cdot b \triangleq \mathbf{lnode}\ (\mathsf{o}\ a \wedge \mathsf{o}\ b)\ (\lambda x.\ \delta_a\ x \cdot b + (\text{if } \mathsf{o}\ a \text{ then } \delta_b\ x \text{ else } \emptyset)),$$

or, concretely as a CoFixpoint in Coq:

```
CoFixpoint concat {n} (a b : lang n) : lang n ≜
  match a, b with
  | lnode b1 k1, lnode b2 k2 ⇒
      lnode (b1 && b2)
        (λ x ⇒ union (concat (k1 x) b) (if b1 then k2 x else empty))
  end.
```

But this definition is rejected by Coq's guardedness checker because the corecursive call to concat appears under a call to union, i.e., concat is primitive recursive "*up-to*" union. One strategy for solving this issue (taken, e.g., by [BBL+17] in the Isabelle/HOL proof assistant) is to register union as a so-called "friendly function" that respects productivity of its arguments. Lacking support for friendly functions in Coq, we take a different approach by defining concatenation as the continuous extension of a monotone fold over finite tries:

Definition 94 (concat)**.** *For languages* $a : \mathsf{lang}_\Sigma$ *and* $b : \mathsf{lang}_\Sigma$*, define the concatenation* $a \cdot b : \mathsf{lang}_\Sigma \triangleq \mathsf{tconcat}^{\mathsf{co}}\ a\ b$*, where:*

$$\mathsf{tconcat} \triangleq \mathsf{fold}_{\mathsf{tlang}_\Sigma}\ (\lambda\_.\ \emptyset)$$
$$(\lambda b.\ \lambda k.\ \lambda l.\ \mathbf{lnode}\ (b \wedge o\ l)(\lambda x.\ k\ x\ l + (\mathit{if}\ b\ \mathit{then}\ \delta_l\ x\ \mathit{else}\ \emptyset))).$$

The Kleene closure operator is afflicted by essentially the same problem. We can attempt to define the closure of $a : \mathit{lang}$ as follows:

$$a^* \triangleq \mathbf{lnode}\ \mathbf{true}\ (\lambda x.\ \delta_a\ x \cdot a^*),$$

or, concretely as a CoFixpoint in Coq:

```
CoFixpoint star {n} (a : lang n) : lang n ≜
  match a with
  | lnode _ k ⇒ lnode true (λ x ⇒ concat (k x) (star a))
  end.
```

But again the definition is rejected because the corecursive call to star appears under a call to concat. We implement the Kleene star is a lazy coiteration (see Section 6.6.2):

Definition 95 (Kleene star)**.** *For language* a : lang_Σ*, define the Kleene closure* a^* : lang_Σ *by:*

$$a^* \triangleq \mathsf{coiter}\ (\lambda b.\ \mathbf{lnode}\ \mathbf{true}\ (\delta_a\ x \cdot b))\ \omega_{\mathbb{N}}.$$

All the operations defined in this section can be extracted to Haskell and used to decide membership of strings over a finite alphabet in regular languages (with an abstract syntax for regular expressions and its interpretation into lang_Σ – see file RE_test.v for example use). We prove finally that they satisfy the Kleene algebra axioms:

Theorem 20 (Kleene Algebra)**.** *The type* lang_Σ *with zero element* $\emptyset$*, one element* ϵ*, and* $+$*,* $\cdot$*, and* * *operations forms a Kleene algebra.*

7 Cotrees

As buds give rise by growth to fresh buds ...

Charles Darwin

Proofs about interaction tree samplers in the Zar system are largely carried out indirectly via the tools of AlgCo through a auxiliary type of coinductive binary trees called *cotrees*. The benefits of this approach are as follows:

1. **Avoiding unneeded generality.** The type of interaction trees is more general than necessary for our purposes, employing higher-order parameterization by an event functor that incurs unnecessary technical friction in proofs.

2. **Separation of constructive and nonconstructive terms.** Cotrees defined by continuous extensions are not necessarily computable. By enforcing a clear separation between the realms of ITrees and cotrees, we ensure that ITrees generated by Zar can always be extracted for execution.

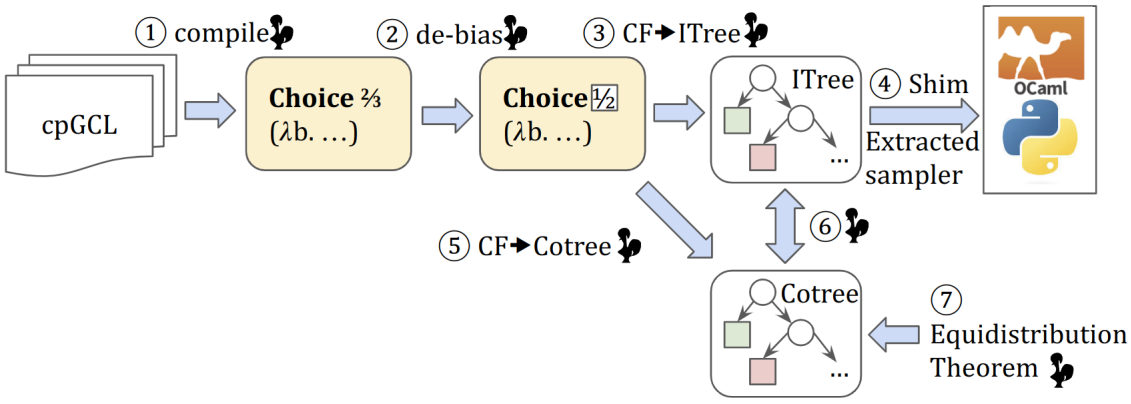


Figure 7.1: Zar compiler pipeline with alternate cotree backend and equidistribution result.

In this chapter, we define the coinductive type of infinitary trees with finite branching factor (*cotrees*) as an algebraic CPO. We then use cotrees to encode samplers for discrete

distributions in the random bit model [VN51, SFRM20b] and show how AlgCo enables weakest pre-expectation [Koz85, MMS96, Kam19] style reasoning about them (Section 7.3), culminating in Theorem 23 showing that sequences of samples are *equidistributed* [BG22] with respect to the weakest pre-expectation semantics of the cotrees that generated them. Figure 7.1 illustrates the Zar architecture extended with (5) compilation from CF trees to cotrees (Section 7.5), (6) relating equivalent constructions of ITrees and cotrees (Section 7.6), and (7) the equidistribution result on cotrees (Section 7.4) which is lifted through (5) to obtain the equivalent result on ITrees.

7.1 Coinductive Trees as an Algebraic CPO

For clarity of presentation we specialize cotrees to the case of coinductive *binary* trees with index type $\mathbb{B}$. In general, the index type must only be finite to ensure compactness of basis elements.

Definition 96 (Cotrees)**.** *Define the type $\mathcal{T}_A^*$ of cotrees with element type A coinductively by the formation rules:*

$$\frac{}{\bot_{\mathcal{T}_A^*} : \mathcal{T}_A^*}\ \textsc{cotree-bot} \qquad \frac{a : A}{\mathbf{coleaf}\ a : \mathcal{T}_A^*}\ \textsc{cotree-leaf} \qquad \frac{t : \mathcal{T}_A^*}{\mathbf{cotau}\ t : \mathcal{T}_A^*}\ \textsc{cotree-tau} \qquad \frac{\forall\, b : \mathbb{B},\ k\, b : \mathcal{T}_A^*}{\mathbf{conode}\ k : \mathcal{T}_A^*}\ \textsc{cotree-node}$$

The definition of cotrees is analogous to that of ITree samplers (Definition 44) but with a special bottom element $\bot_{\mathcal{T}_A^*}$ standing for the "undefined" cotree. '**coleaf** a' encodes a sampler that produces sample $a : A$, and '**conode** k' encodes a binary choice between subtrees 'k **true**' and 'k **false**'. As alluded to in Section 6.1, the **cotau** constructor is not strictly necessary for any of the definitions in this chapter, but we include it to enable a straightforward injection of ITrees into the type of cotrees for the purpose of defining weakest pre-expectation semantics on ITrees (see Section 7.6 for details). Figure 7.2

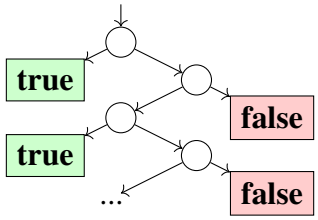


Figure 7.3: Illustration of cotree encoding of Bernoulli($\frac{2}{3}$) distribution from Figure 7.2.

shows a cotree analogue of the Bernoulli($\frac{2}{3}$) ITree from Figure 4.5, illustrated in Figure 7.3.

$$ct_{\frac{2}{3}} \triangleq \textbf{conode}\ (\lambda b_0.\ \text{if}\ b_0\ \text{then}\ \textbf{coleaf true}\ \text{else}\ \textbf{conode}\ (\lambda b_1.\ \text{if}\ b_1\ \text{then}\ \textbf{coleaf false}\ \text{else}\ \textbf{cotau}\ ct_{\frac{2}{3}}))$$

Figure 7.2: Cotree term corresponding to the ITree in Figure 4.5.

Cotrees are ordered by a straightforward structural prefix relation:

Definition 97 (Cotree order)**.** *Define* $\sqsubseteq_{\mathcal{T}_A^*} : \mathcal{T}_A^* \to \mathcal{T}_A^* \to \mathbb{P}$ *coinductively by the inference rules:*

$$\frac{l : \mathcal{T}_A^*}{\bot_{\mathcal{T}_A^*} \sqsubseteq_{\mathcal{T}_A^*} l}\ \sqsubseteq_{\mathcal{T}_A^*}\text{-BOT} \qquad \frac{a : A}{\textbf{coleaf}\ a \sqsubseteq_{\mathcal{T}_A^*} \textbf{coleaf}\ a}\ \sqsubseteq_{\mathcal{T}_A^*}\text{-LEAF} \qquad \frac{t_1 \sqsubseteq_{\mathcal{T}_A^*} t_2}{\textbf{cotau}\ t_1 \sqsubseteq_{\mathcal{T}_A^*} \textbf{cotau}\ t_2}\ \sqsubseteq_{\mathcal{T}_A^*}\text{-TAU}$$

$$\frac{\forall\, b : \mathbb{B},\ f\ b \sqsubseteq_{\mathcal{T}_A^*} g\ b}{\textbf{conode}\ f \sqsubseteq_{\mathcal{T}_A^*} \textbf{conode}\ g}\ \sqsubseteq_{\mathcal{T}_A^*}\text{-NODE}$$

A compact basis for cotrees is given by a corresponding ordered type of finite binary trees:

Definition 98 (Finite trees)**.** *Define the type $\mathcal{T}_A$ of finite binary trees with element type A inductively by the formation rules:*

$$\frac{}{\bot_{\mathcal{T}_A} : \mathcal{T}_A}\ \text{TREE-BOT} \qquad \frac{a : A}{\mathbf{leaf}\ a : \mathcal{T}_A}\ \text{TREE-LEAF} \qquad \frac{t : \mathcal{T}_A}{\mathbf{tau}\ t : \mathcal{T}_A}\ \text{TREE-TAU} \qquad \frac{\forall\, b : \mathbb{B},\ k\ b : \mathcal{T}_A}{\mathbf{node}\ k : \mathcal{T}_A}\ \text{TREE-NODE}$$

With prefix order:

Definition 99 (Finite tree order)**.** *Define $\sqsubseteq_{\mathcal{T}_A} : \mathcal{T}_A \to \mathcal{T}_A \to \mathbb{P}$ inductively by the inference rules:*

$$\frac{t : \mathcal{T}_A}{\bot_{\mathcal{T}_A} \sqsubseteq_{\mathcal{T}_A} t}\ \sqsubseteq_{\mathcal{T}_A}\text{-BOT} \qquad \frac{a : A}{\mathbf{leaf}\ a \sqsubseteq_{\mathcal{T}_A} \mathbf{leaf}\ a}\ \sqsubseteq_{\mathcal{T}_A}\text{-LEAF} \qquad \frac{t_1 \sqsubseteq_{\mathcal{T}_A} t_2}{\mathbf{tau}\ t_1 \sqsubseteq_{\mathcal{T}_A} \mathbf{tau}\ t_2}\ \sqsubseteq_{\mathcal{T}_A}\text{-TAU} \qquad \frac{\forall\, b : \mathbb{B},\ f\ b \sqsubseteq_{\mathcal{T}_A} g\ b}{\mathbf{node}\ f \sqsubseteq_{\mathcal{T}_A} \mathbf{node}\ g}\ \sqsubseteq_{\mathcal{T}_A}\text{-NODE}$$

Remark 15 ($\mathcal{T}_A$ is compact)*.* For any type A, the elements of $\mathcal{T}_A$ are finite and are thus compact. Any function with domain $\mathcal{T}_A$ is therefore automatically continuous (Lemma 4) and, importantly, can be reasoned about by induction over the domain $\mathcal{T}_A$.

To show that $\mathcal{T}_A^*$ is an algebraic CPO, it only remains to show that $\mathcal{T}_A$ is dense in $\mathcal{T}_A^*$. The inclusion map $\mathsf{incl}_{\mathcal{T}_A,\mathcal{T}_A^*} : \mathcal{T}_A \to \mathcal{T}_A^*$ injects finite trees into $\mathcal{T}_A^*$, and $\mathsf{idl}_{\mathcal{T}_A,\mathcal{T}_A^*} : \mathcal{T}_A^* \to \mathbb{N} \to \mathcal{T}_A$ generates convergent chains of finite approximations of cotrees.

Definition 100 ($\mathsf{incl}_{\mathcal{T},\mathcal{T}^*}$)**.** *For type A and $t : \mathcal{T}_A$, define $\mathsf{incl}_{\mathcal{T}_A,\mathcal{T}_A^*}\ l : \mathcal{T}_A^*$ by induction on l:*

$\mathsf{incl}_{\mathcal{T}_A,\mathcal{T}_A^*} : \mathcal{T}_A \to \mathcal{T}_A^*$

$$\begin{aligned}
\bot_{\mathcal{T}_A} &\triangleq \bot_{\mathcal{T}_A^*} \\
\mathbf{leaf}\ a &\triangleq \mathbf{coleaf}\ a \\
\mathbf{tau}\ t &\triangleq \mathbf{cotau}\ (\mathsf{incl}_{\mathcal{T}_A,\mathcal{T}_A^*}\ t) \\
\mathbf{node}\ k &\triangleq \mathbf{conode}\ (\mathsf{incl}_{\mathcal{T}_A,\mathcal{T}_A^*} \circ k)
\end{aligned}$$

Definition 101 ($\mathsf{idl}_{\mathcal{T},\mathcal{T}^*}$)**.** *For type A, $l : \mathcal{T}_A^*$, and $n : \mathbb{N}$, define $\mathsf{idl}_{\mathcal{T}_A,\mathcal{T}_A^*}\ l\ n : \mathcal{T}_A$ by induction on n:*

$\mathsf{idl}_{\mathcal{T}_A,\mathcal{T}_A^*} : \mathcal{T}_A^* \to \mathbb{N} \to \mathcal{T}_A^*$			
_	**O**	$\triangleq$	$\bot_{\mathcal{T}_A}$
$\bot_{\mathcal{T}_A^*}$	_	$\triangleq$	$\bot_{\mathcal{T}_A}$
coleaf a	(**S** n)	$\triangleq$	**leaf** a
(**cotau** t)	(**S** n)	$\triangleq$	**tau** ($\mathsf{idl}_{\mathcal{T}_A,\mathcal{T}_A^*}\ t\ n$)
(**conode** k)	(**S** n)	$\triangleq$	**node** ($\lambda b.\ \mathsf{idl}_{\mathcal{T}_A,\mathcal{T}_A^*}\ (k\ b)\ n$)

It follows that the type $\mathcal{T}_A$ of finite binary trees is dense in the type $\mathcal{T}_A^*$ of cotrees for any type A.

Remark 16 ($\mathcal{T}_A$ is dense in $\mathcal{T}_A^*$)*.* Let A be a type. Then, for all $a : \mathcal{T}_A^*$,

$$\mathsf{idl}\ a \text{ is an } \omega\text{-chain, } \textit{and}$$

$$\mathsf{sup}\ (\mathsf{incl} \circ \mathsf{idl}\ a) = a.$$

We also have $\bot_{\mathcal{T}_A^*} \sqsubseteq l$ for all $l : \mathcal{T}_A^*$, and thus:

Remark 17 ($\mathcal{T}_A^*$ is a pointed algebraic CPO with basis $\mathcal{T}_A$).

The order relation on cotrees (Definition 97) is carefully chosen to allow the following extensionality axiom entailing propositional equality from order equivalence (see Section 6.6.1 for discussion of such extensionality axioms for coinductive types):

Axiom 7 (Cotree extensionality)**.** *Let A be a type. Then,*

$$\forall\ l_1\ l_2 : \mathcal{T}_A^*,\ l_1 \simeq_{\mathcal{T}_A^*} l_2 \Rightarrow l_1 = l_2.$$

7.2 Cofolds Over Cotrees

Cofolds over cotrees and their computation rules are derived analogously to streams (Section 6.7.1) with respect to the fold operator:

Definition 102 ($\mathsf{fold}_{\mathcal{T}_A}$)**.** *For type A, ordered type B, $z : B$, $f : A \to B$, $g : B \to B$, $h : (\mathbb{B} \to B) \to B$, and $t : \mathcal{T}_A$, define $\mathsf{fold}_{\mathcal{T}_A}\ z\ f\ g\ h\ t : B$ by induction on t:*

$\mathsf{fold}_{\mathcal{T}_A}\ z\ f\ g\ h : \mathcal{T}_A \to B$		
$\bot_{\mathcal{T}_A}$	$\triangleq$	z
leaf a	$\triangleq$	$f\ a$
tau t	$\triangleq$	$g\ (\mathsf{fold}_{\mathcal{T}_A}\ z\ f\ g\ h\ t)$
node k	$\triangleq$	$h\ (\mathsf{fold}_{\mathcal{T}_A}\ z\ f\ g\ h \circ k)$

Definition 103 ($\mathsf{cofold}_{\mathcal{T}_A^*}$)**.** *For type A, pointed type B, $f : A \to B$, $g : B \to B$, and $h : (\mathbb{B} \to B) \to B$, define $\mathsf{cofold}_{\mathcal{T}_A^*}\ f\ g\ h : \mathcal{T}_A^* \to B$ by:*

$$\mathsf{cofold}_{\mathcal{T}_A^*}\ f\ g\ h \triangleq (\mathsf{fold}_{\mathcal{T}_A}\ \bot_B\ f\ g\ h)^{\mathsf{co}},$$

or when B has a top element $\top_B$, define $\hat{\mathsf{cofold}}_{\mathcal{T}_A^*}\ f\ g\ h : \mathcal{L}_A^* \to B$:

$$\hat{\mathsf{cofold}}_{\mathcal{T}_A^*}\ f\ g\ h \triangleq (\mathsf{fold}_{\mathcal{T}_A}\ \top_B\ f\ g\ h)^{\hat{\mathsf{co}}}.$$

To illustrate with a concrete example, let us define the functorial map operator for cotrees as a cofold:

Definition 104 ($\mathsf{map}_{\mathcal{T}_A^*}$)**.** *For types A and B, and $f : A \to B$, define $\mathsf{map}_{\mathcal{T}_A^*}\ f : \mathcal{T}_A^* \to \mathcal{T}_B^*$ by:*

$$\mathsf{map}_{\mathcal{T}_A}\ f \triangleq \mathsf{cofold}_{\mathcal{T}_A}\ (\mathbf{coleaf} \circ f)\ \mathbf{cotau}\ \mathbf{conode}.$$

Proofs of continuity for cofolds can be simplified via the following general lemma:

Lemma 16 ((Co)continuous cofold for cotrees)**.** *Let A be a type, B an ordered type, $z : B$, $f : A \to B$, $g : B \to B$ monotone, and $h : (\mathbb{B} \to B) \to B$ monotone. Then,*

$$(\forall\ t : \mathcal{T}_A,\ z \sqsubseteq \mathsf{fold}\ z\ f\ g\ h\ t) \Rightarrow \mathsf{fold}\ z\ f\ g\ h \text{ is monotone},$$

$$(\forall\ t : \mathcal{T}_A,\ \mathsf{fold}\ z\ f\ g\ h\ t \sqsubseteq z) \Rightarrow \mathsf{fold}\ z\ f\ g\ h \text{ is antimonotone, and}$$

$$\mathsf{cofold}_{\mathcal{T}_A^*}\ f \text{ is continuous and } \hat{\mathsf{cofold}}_{\mathcal{T}_A^*}\ f \text{ is cocontinuous}.$$

Applying Lemma 16 to $\mathsf{map}_{\mathcal{T}^*}$, we see that $\mathsf{map}_{\mathcal{T}^*}$ is continuous because the **cotau** and **conode** constructors are monotone.

Cofold computation Computation rules for cofolds can be derived from the following generic lemma:

Lemma 17 ($\mathsf{cofold}_{\mathcal{T}^*_A}$ computation for cotrees)**.** *Let A be a type, B a pointed CPO, $f : A \to B$, $g : B \to B$, and $h : (\mathbb{B} \to B) \to B$ such that g and h are continuous. Then,*

$$\begin{aligned}
\mathsf{cofold}_{\mathcal{T}^*_A}\ f\ g\ h\ \bot_{\mathcal{T}^*_A} &\simeq \bot_B \\
\mathsf{cofold}_{\mathcal{T}^*_A}\ f\ g\ h\ (\mathbf{coleaf}\ a) &\simeq f\ a \\
\mathsf{cofold}_{\mathcal{T}^*_A}\ f\ g\ h\ (\mathbf{cotau}\ t) &\simeq g\ (\mathsf{cofold}_{\mathcal{T}^*_A}\ f\ g\ h\ t) \\
\mathsf{cofold}_{\mathcal{T}^*_A}\ f\ g\ h\ (\mathbf{conode}\ k) &\simeq h\ (\mathsf{cofold}_{\mathcal{T}^*_A}\ f\ g\ h\ \circ k).
\end{aligned}$$

By applying Lemma 17 to our running example, we derive the following computation rules for $\mathsf{map}_{\mathcal{T}^*}$:

Corollary 2 ($\mathsf{map}_{\mathcal{T}^*}$ computation)**.**

$$\begin{aligned}
\mathsf{map}_{\mathcal{T}^*_A}\ f\ \bot_{\mathcal{T}^*_A} &= \bot_{\mathcal{T}^*_A} \\
\mathsf{map}_{\mathcal{T}^*_A}\ f\ (\mathbf{coleaf}\ a) &= \mathbf{coleaf}\ (f\ a) \\
\mathsf{map}_{\mathcal{T}^*_A}\ f\ (\mathbf{cotau}\ t) &= \mathbf{cotau}\ (\mathsf{map}_{\mathcal{T}^*_A}\ f\ t) \\
\mathsf{map}_{\mathcal{T}^*_A}\ f\ (\mathbf{conode}\ k) &= \mathbf{conode}\ (\mathsf{map}_{\mathcal{T}^*_A}\ f \circ k).
\end{aligned}$$

Lemma 17 guarantees partial correctness (i.e, correctness for terminating executions) of the extraction primitive in Figure 7.4 for continuous g and h (see Section 6.7.5 for related discussion on partial correctness of extracted cofolds).

Essential Cofolds We present a collection of useful cofolds over cotrees and derive their computation rules from Lemma 17, beginning with the monadic bind operator for cotrees:

Definition 105 (Cotree monadic bind)**.** *For types A and B, $t : \mathcal{T}^*_A$, and $f : A \to \mathcal{T}^*_B$, define $t \gg\!= f : \mathcal{T}^*_B \triangleq (\mathsf{tbind}\ f)^{\mathsf{co}}\ t$, where:*

$$\mathsf{tbind}\ f \triangleq \mathsf{fold}_{\mathcal{T}_A}\ \bot_{\mathcal{T}^*_B}\ k\ \mathbf{cotau}\ \mathbf{conode}$$

```
cofold =
  \ o p f g h t ->
    case t of
      Cobot -> bot o p
      Coleaf a -> f a
      Cotau t' -> g (cofold o p f g h t')
      Conode k -> g (cofold o p f g h . k)
```

Figure 7.4: Haskell extraction primitive for $\mathsf{cofold}_{\mathcal{T}^*}$.

with computation rules:

$$
\begin{array}{lcll}
\bot_{\mathcal{T}^*_A} & \gg\!= & f & = \bot_{\mathcal{T}^*_A} \\
\textbf{coleaf}\ a & \gg\!= & f & = f\ a \\
\textbf{cotau}\ t & \gg\!= & f & = \textbf{cotau}\ (t \gg\!= f) \\
\textbf{conode}\ k & \gg\!= & f & = \textbf{conode}\ (\lambda x.\ k\ x \gg\!= f).
\end{array}
$$

Remark 18 ($\mathcal{T}^*$ is a monad). The type constructor $\mathcal{T}^*$ is a monad with monadic return given by the **leaf** constructor and bind given by $\gg\!=$ (Definition 105). That is,

leaf is the right identity for $t : \mathcal{T}^*_A$:

$$t \gg\!= \textbf{leaf} = t,$$

and '$\gg\!=$' is associative for $t : \mathcal{T}^*_A$, $f : A \to \mathcal{T}^*_B$, $g : B \to \mathcal{T}^*_C$:

$$(t \gg\!= f) \gg\!= g = t \gg\!= (\lambda x.\ f\ x \gg\!= g).$$

Existential quantification over cotrees is given by the continuous predicate:

Definition 106 ($\exists_P^{\text{co}}$)**.** *For type A and predicate $P : A \to \mathbb{P}$, define $\exists_P^{\text{co}} : \mathcal{T}_A^* \to \mathbb{P}$ where:*

$$\exists_P \triangleq \mathsf{fold}_{\mathcal{T}_A}\ \bot\ P\ (\lambda g : \mathbb{B} \to \mathbb{P}.\ \exists\, b.\ g\ b)$$

with derived introduction and elimination rules:

$\exists_P^{\text{co}}$-INTRO-LEAF
$$\frac{P\ a}{\exists_P^{\text{co}}\ (\mathbf{coleaf}\ a)}$$

$\exists_P^{\text{co}}$-INTRO-TAU
$$\frac{\exists_P^{\text{co}}\ t}{\exists_P^{\text{co}}\ (\mathbf{cotau}\ t)}$$

$\exists_P^{\text{co}}$-INTRO-NODE
$$\frac{b : \mathbb{B} \qquad \exists_P^{\text{co}}\ (k\ b)}{\exists_P^{\text{co}}\ (\mathbf{conode}\ k)}$$

$\exists_P^{\text{co}}$-ELIM-BOT
$$\frac{\exists_P^{\text{co}}\ \bot_{\mathcal{T}_A^*}}{\bot}$$

$\exists_P^{\text{co}}$-ELIM-LEAF
$$\frac{\exists_P^{\text{co}}\ (\mathbf{coleaf}\ a)}{P\ a}$$

$\exists_P^{\text{co}}$-ELIM-TAU
$$\frac{\exists_P^{\text{co}}\ (\mathbf{cotau}\ t)}{\exists_P^{\text{co}}\ t}$$

$\exists_P^{\text{co}}$-ELIM-NODE
$$\frac{\exists_P^{\text{co}}\ (\mathbf{conode}\ k)}{\exists\, b : \mathbb{B}.\ \exists_P^{\text{co}}\ (k\ b)}$$

The reader may notice that the expression '$\exists b.\ g\ b$' above could be replaced with the arguably simpler expression 'g **false** $\vee$ g **true**'. This is an artifact of our specialization to *binary* cotrees; in general the index type for subtrees can be any finite type.

Universal quantification over cotrees is given by the cocontinuous predicate:

Definition 107 ($\forall_P^{\hat{\text{co}}}$)**.** *For type A and predicate $P : A \to \mathbb{P}$, define $\forall_P^{\hat{\text{co}}} : \mathcal{T}_A^* \to \mathbb{P}$ where:*

$$\forall_P \triangleq \mathsf{fold}_{\mathcal{T}_A}\ \top\ P\ (\lambda g : \mathbb{B} \to \mathbb{P}.\ \forall\, b.\ g\ b)$$

with introduction and elimination rules:

$\forall_P^{\hat{\text{co}}}$-INTRO-BOT
$$\frac{}{\forall_P^{\hat{\text{co}}}\ \bot_{\mathcal{T}_A^*}}$$

$\forall_P^{\hat{\text{co}}}$-INTRO-LEAF
$$\frac{P\ a}{\forall_P^{\hat{\text{co}}}\ (\mathbf{coleaf}\ a)}$$

$\forall_P^{\hat{\text{co}}}$-INTRO-TAU
$$\frac{\forall_P^{\hat{\text{co}}}\ t}{\forall_P^{\hat{\text{co}}}\ (\mathbf{cotau}\ t)}$$

$\forall_P^{\hat{\text{co}}}$-INTRO-NODE
$$\frac{\forall\, b : \mathbb{B}.\ \forall_P^{\hat{\text{co}}}\ (k\ b)}{\forall_P^{\hat{\text{co}}}\ (\mathbf{conode}\ k)}$$

$\forall_P^{\hat{\text{co}}}$-ELIM-LEAF
$$\frac{\forall_P^{\hat{\text{co}}}\ (\mathbf{coleaf}\ a)}{P\ a}$$

$\forall_P^{\hat{\text{co}}}$-ELIM-TAU
$$\frac{\forall_P^{\hat{\text{co}}}\ (\mathbf{cotau}\ t)}{\forall_P^{\hat{\text{co}}}\ t}$$

$\forall_P^{\hat{\text{co}}}$-ELIM-NODE
$$\frac{b : \mathbb{B} \qquad \forall_P^{\hat{\text{co}}}\ (\mathbf{conode}\ k)}{\forall_P^{\hat{\text{co}}}\ (k\ b)}$$

As for Definition 106, we use a universal quantifier to accommodate the possibility of index types other than $\mathbb{B}$.

7.3 Weakest Pre-Expectations for Cotrees

To reason formally about cotree samplers, we define a variant of the *weakest pre-expectation* (wp) semantics (originally due to Kozen [Koz85]), adapted from its application to the *probabilistic Guarded Command Language* (pGCL) [MM05].

An *expectation* is a function $f : A \to \mathbb{R}^{\infty}_{\geq 0}$ mapping elements of sample space A to the nonnegative extended reals. The purpose of wp is to compute expected values of expectations over cotrees (i.e., integrating over the probability densities encoded by them):

Definition 108 ($\mathsf{wp}_{\mathcal{T}^*}$)**.** *For type A and expectation $f : A \to \mathbb{R}^{\infty}_{\geq 0}$, define* $\mathsf{wp}_{\mathcal{T}^*_A}\ f : \mathcal{T}^*_A \to \mathbb{R}^{\infty}_{\geq 0} \triangleq (\mathsf{wp}_{\mathcal{T}_A}\ f)^{\mathsf{co}}$*, where:*

$$\mathsf{wp}_{\mathcal{T}_A}\ f \triangleq \mathsf{fold}_{\mathcal{T}_A}\ 0\ f\ \mathsf{id}_{\mathbb{R}^{\infty}_{\geq 0}}\ (\lambda g : \mathbb{B} \to \mathbb{R}^{\infty}_{\geq 0}.\ \frac{g\ \mathbf{true} + g\ \mathbf{false}}{2})$$

with computation rules:

$$\begin{aligned}
\mathsf{wp}_{\mathcal{T}^*_A}\ f\ \bot_{\mathcal{T}^*_A} &= 0 \\
\mathsf{wp}_{\mathcal{T}^*_A}\ f\ (\mathbf{coleaf}\ a) &= f\ a \\
\mathsf{wp}_{\mathcal{T}^*_A}\ f\ (\mathbf{cotau}\ t) &= \mathsf{wp}_{\mathcal{T}^*_A}\ f\ t \\
\mathsf{wp}_{\mathcal{T}^*_A}\ f\ (\mathbf{conode}\ k) &= \frac{\mathsf{wp}_{\mathcal{T}^*_A}\ f\ (k\ \mathbf{true}) + \mathsf{wp}_{\mathcal{T}^*_A}\ f\ (k\ \mathbf{false})}{2}.
\end{aligned}$$

$\mathsf{id}_{\mathbb{R}^{\infty}_{\geq 0}}$ denotes the identity mapping on $\mathbb{R}^{\infty}_{\geq 0}$. $\mathsf{wp}_{\mathcal{T}^*_A}$ can be used to express the probability of a given event $Q : A \to \mathbb{P}$ over the distribution encoded by cotree $t : \mathcal{T}^*_A$ via the expected value of the indicator function $[Q]$ given by $\mathsf{wp}_{\mathcal{T}^*_A}\ [Q]\ t$. For example, the probability that $t_{\frac{2}{3}}$ produces the value **true** is given by $\mathsf{wp}\ [\lambda x.\ x = \mathbf{true}]\ t_{\frac{2}{3}} = \frac{2}{3}$. Technically, $\mathsf{wp}_{\mathcal{T}^*_A}\ [Q]\ t$ denotes the probability of *terminating with a sample satisfying Q*, and does not account for executions which produce no sample at all. For more flexibility in reasoning about nontermination, we also define a "liberal" variant $\mathsf{wlp}_{\mathcal{T}^*}$:

Definition 109 ($\mathsf{wlp}_{\mathcal{T}^*}$)**.** *For type A and expectation $f : A \to \mathbb{R}^{\leq 1}_{\geq 0}$, define $\mathsf{wlp}_{\mathcal{T}^*_{\mathsf{A}}}\ f : \mathcal{T}^*_A \to \mathbb{R}^{\leq 1}_{\geq 0} \triangleq (\mathsf{wlp}_{\mathcal{T}_{\mathsf{A}}}\ f)^{\mathsf{co}}$, where:*

$$\mathsf{wlp}_{\mathcal{T}_{\mathsf{A}}}\ f \triangleq \mathsf{fold}_{\mathcal{T}_{\mathsf{A}}}\ 1\ f\ \mathsf{id}_{\mathbb{R}^{\leq 1}_{\geq 0}}\ (\lambda g : \mathbb{B} \to \mathbb{R}^{\leq 1}_{\geq 0}.\ \frac{g\ \mathbf{true} + g\ \mathbf{false}}{2})$$

with computation rules:

$$\begin{aligned}
\mathsf{wlp}_{\mathcal{T}^*_{\mathsf{A}}}\ f\ \bot_{\mathcal{T}^*_A} &= 1 \\
\mathsf{wlp}_{\mathcal{T}^*_{\mathsf{A}}}\ f\ (\mathbf{coleaf}\ a) &= f\ a \\
\mathsf{wlp}_{\mathcal{T}^*_{\mathsf{A}}}\ f\ (\mathbf{cotau}\ t) &= \mathsf{wlp}_{\mathcal{T}^*_{\mathsf{A}}}\ f\ t \\
\mathsf{wlp}_{\mathcal{T}^*_{\mathsf{A}}}\ f\ (\mathbf{conode}\ k) &= \frac{\mathsf{wlp}_{\mathcal{T}^*_{\mathsf{A}}}\ f\ (k\ \mathbf{true}) + \mathsf{wlp}_{\mathcal{T}^*_{\mathsf{A}}}\ f\ (k\ \mathbf{false})}{2}.
\end{aligned}$$

$\mathsf{wlp}_{\mathcal{T}^*_{\mathsf{A}}}\ [Q]\ t$ denotes the probability of producing a sample satisfying Q *plus the probability of divergence*. Note that $\mathsf{wlp}_{\mathcal{T}^*}$ is only defined for expectations bounded above by 1 and thus is primarily used for calculation of probabilities. Also note that $\mathsf{wlp}_{\mathcal{T}^*}$ is *cocontinuous*, whereas $\mathsf{wp}_{\mathcal{T}^*}$ is continuous.

Conditional Weakest Pre-expectations cpGCL programs are compiled to cotrees of type $\mathcal{T}^*_{\mathbf{1}+\Sigma}$ (see Section 7.5), where $\mathbf{1}$ on the LHS of the sum type is used to encode observation failure. I.e., a terminating execution of the process encoded by $t : \mathcal{T}^*_{\mathbf{1}+\Sigma}$ produces either the value $\mathsf{inl}\ ()$ denoting failed observation, or $\mathsf{inr}\ \sigma$ for some terminal program state $\sigma : \Sigma$. We can thus recreate the generalized wp_{b} semantics described in Section 3.3 of $t : \mathcal{T}^*_{\mathbf{1}+\Sigma}$ as $\mathsf{wp}_{\mathcal{T}^*_{\mathbf{1}+\Sigma}}\ ([\mathsf{b}] + f)\ t$, (this connection is made explicit in Section 7.5), and likewise for wlp. The cwp semantics of cotrees is then defined analogously to Definitions 2 and 40:

Definition 110 ($\mathsf{cwp}_{\mathcal{T}^*}$)**.** *For type A, $t : \mathcal{T}^*_{\mathbf{1}+A}$, and $f : A \to \mathbb{R}^{\infty}_{\geq 0}$, define $\mathsf{cwp}_{\mathcal{T}^*_{\mathsf{A}}}\ f\ t : \mathbb{R}^{\infty}_{\geq 0}$ by:*

$$\mathsf{cwp}_{\mathcal{T}^*_{\mathsf{A}}}\ f\ t \triangleq \frac{\mathsf{wp}_{\mathcal{T}^*_{\mathbf{1}+\mathsf{A}}}\ (\mathbf{0} + f)\ t}{\mathsf{wlp}_{\mathcal{T}^*_{\mathbf{1}+\mathsf{A}}}\ (\mathbf{0} + \mathbf{1})\ t}.$$

Healthiness Conditions The following lemma shows a fundamental connection between $\mathsf{wp}_{\mathcal{T}^*}$ and monadic bind that is especially relevant when reasoning about cotree samplers compiled from probabilistic programs with sequenced commands:

Lemma 18 ($\mathsf{wp}_{\mathcal{T}^*}$ bind)**.** *Let A and B be types, $f : B \to \mathbb{R}^{\infty}_{\geq 0}$, $t : \mathcal{T}^*_A$, and $k : A \to \mathcal{T}^*_B$. Then,*

$$\mathsf{wp}_{\mathcal{T}^*_B}\, f\, (t \gg\!= k) = \mathsf{wp}_{\mathcal{T}^*_A}\, (\mathsf{wp}_{\mathcal{T}^*_B} f \circ k)\, t.$$

An analogous lemma holds for $\mathsf{wlp}_{\mathcal{T}^*}$. We further validate $\mathsf{wp}_{\mathcal{T}^*}$ and $\mathsf{wlp}_{\mathcal{T}^*}$ by proving several healthiness conditions, including Markov's inequality [Kam19]. All proofs are carried out with the machinery of AlgCo from Chapter 6.

Lemma 19 ($\mathsf{wp}_{\mathcal{T}^*}$ is strict and $\mathsf{wlp}_{\mathcal{T}^*}$ is co-strict)**.** *For any type A,*

$$\mathsf{wp}_{\mathcal{T}^*_A}\, \mathbf{0} = \mathbf{0}, \qquad\qquad \mathsf{wlp}_{\mathcal{T}^*_A}\, \mathbf{1} = \mathbf{1}.$$

Lemma 20 ($\mathsf{wp}_{\mathcal{T}^*}$ is linear)**.** *Let A be a type, $c : \mathbb{R}^{\infty}_{\geq 0}$, $f, g : A \to \mathbb{R}^{\infty}_{\geq 0}$, and $t : \mathcal{T}^*_A$. Then,*

$$\mathsf{wp}_{\mathcal{T}^*_A}\, (c \cdot f + g)\, t = c \cdot \mathsf{wp}_{\mathcal{T}^*_A}\, f\, t + \mathsf{wp}_{\mathcal{T}^*_A}\, f\, t.$$

Lemma 21 ($\mathsf{wp}_{\mathcal{T}^*}$ and $\mathsf{wlp}_{\mathcal{T}^*}$ are monotone)**.** *Let A be a type, $f, g : A \to \mathbb{R}^{\infty}_{\geq 0}$, and $t : \mathcal{T}^*_A$. Then,*

$$(\forall\, a : A,\ f\, a \leq g\, a) \Rightarrow \mathsf{wp}_{\mathcal{T}^*_A}\, f\, t \leq \mathsf{wp}_{\mathcal{T}^*_A}\, f\, t,$$

and if f and g are bounded above by 1*,*

$$(\forall\, a : A,\ f\, a \leq g\, a) \Rightarrow \mathsf{wlp}_{\mathcal{T}^*_A}\, f\, t \leq \mathsf{wlp}_{\mathcal{T}^*_A}\, f\, t.$$

Lemma 22 (Invariant sum)**.** *Let A be a type, $f : A \to \mathbb{R}^{\leq 1}_{\geq 0}$, and $t : \mathcal{T}^*_A$. Then,*

$$\mathsf{wp}_{\mathcal{T}^*_A}\, f\, t + \mathsf{wlp}_{\mathcal{T}^*_A}\, (\mathbf{1} - f)\, t = 1.$$

Theorem 21 (Markov's inequality)**.** *Let A be a type, $f : A \to \mathbb{R}^{\infty}_{\geq 0}$, $t : \mathcal{T}^*_A$, and $a : \mathbb{R}_{\geq 0}$. Then,*

$$\mathsf{wp}_{\mathcal{T}^*_A}\, [f \geq a]\, t \leq \frac{\mathsf{wp}_{\mathcal{T}^*_A}\, f\, t}{a}.$$

7.4 Coinductive Measure

As described in Section 5.2, we can view a cotree sampler as an encoding of a partial function mapping infinite bitstreams (elements of the *Cantor space* 2^ω) to elements of the sample space. Given $t : \mathcal{T}_A^*$, we let $f_t : 2^\omega \rightharpoonup A$ denote the function induced by t that either diverges on a given bitstream or produces a sample x by starting from the root of t and using the bits of the stream to guide traversal (e.g., taking the left subtree on 1 and the right on 0) until reaching a leaf containing x.

The *preimage* $f_t^{-1}(Q)$ of an event $Q : A \to \mathbb{P}$ under sampler function f_t is the subset of bitstreams in 2^ω sent by f_t to samples in Q. We represent subsets of 2^ω as cotrees of type $\mathcal{T}^*_{\mathcal{L}_\mathbb{B}}$ encoding countable unions of *basic sets* in 2^ω. Basic sets are encoded by finite bitstrings, where bitstring $b : \mathcal{L}_\mathbb{B}$ denotes the set $\{s : 2^\omega \mid b \sqsubseteq s\}$ of all bitstreams sharing prefix b. We further require that all bitstrings appearing in a cotree set be pairwise incomparable, i.e., disjoint. We let Σ^0_1 denote the class of countable unions of basic sets, and show how to compute cotree representations of Σ^0_1 preimages of events via a continuous extension in Definition 113.

Under this view, we re-cast the task of inferring the probability of event $Q : A \to \mathbb{P}$ with respect to sampler $t : \mathcal{T}_A^*$ to that of computing the *measure* of $f_t^{-1}(Q) \subseteq 2^\omega$, where the measure of a bitstring bs is equal to $\frac{1}{2^{(\mathit{length}\ bs)}}$, and the measure of a Σ^0_1 cotree (Definition 117) is the sum of the measures of its constituent bitstrings (so long as they are pairwise disjoint). We prove that the probability of any event Q according to the $\mathsf{wp}_{\mathcal{T}_A^*}$ semantics of sampler t coincides with the measure of its preimage under f_t (Lemma 25), and then use this to prove that sequences of samples generated from t are equidistributed with respect to its $\mathsf{wp}_{\mathcal{T}_A^*}$ semantics (Theorem 23).

Computing Preimages Preimages of cotrees are defined by the composition of two continuous extensions $\mathsf{lang}_{\mathcal{T}_A^*}$ and $\mathsf{filter}_{\mathcal{T}_A^*}$. We begin with $\mathsf{lang}_{\mathcal{T}_A^*}$, which indiscriminately

(not with respect to any predicate) computes the subset of 2^ω sent by sampler t to *any leaf at all* (i.e., the preimage $f_t^{-1}(A)$ of the entire sample space A, or, the *language* of t).

Definition 111 ($\mathsf{lang}_{\mathcal{T}^*_{\mathsf{A}}}$)**.** *For type A, define* $\mathsf{lang}_{\mathcal{T}^*_{\mathsf{A}}} : \mathcal{T}^*_A \to \mathcal{T}^*_{\mathcal{L}_{\mathbb{B}}} \triangleq \mathsf{lang}_{\mathcal{T}_{\mathsf{A}}}{}^{\mathsf{co}}$*, where:*

$$\mathsf{lang}_{\mathcal{T}_{\mathsf{A}}} \triangleq \mathsf{fold}_{\mathcal{T}_{\mathsf{A}}}\ \bot_{\mathcal{T}^*_{\mathcal{L}_{\mathbb{B}}}}\ (\lambda\_.\ \mathbf{coleaf}\ \mathbf{nil})\ \mathsf{id}_{\mathcal{T}^*_{\mathcal{L}_{\mathbb{B}}}}$$
$$(\lambda g.\ \mathbf{conode}\ (\lambda b.\ \mathsf{map}_{\mathcal{T}^*_{\mathsf{A}}}\ (\mathbf{cons}\ b)\ (g\ b)))$$

with computation rules:

$$\begin{aligned}
\mathsf{lang}_{\mathcal{T}^*_{\mathsf{A}}}\ f\ \bot_{\mathcal{T}^*_A} &= \bot_{\mathcal{T}^*_{\mathcal{L}_{\mathbb{B}}}} \\
\mathsf{lang}_{\mathcal{T}^*_{\mathsf{A}}}\ f\ (\mathbf{coleaf}\ a) &= \mathbf{coleaf}\ \mathbf{nil} \\
\mathsf{lang}_{\mathcal{T}^*_{\mathsf{A}}}\ f\ (\mathbf{cotau}\ t) &= \mathsf{lang}_{\mathcal{T}^*_{\mathsf{A}}}\ f\ t \\
\mathsf{lang}_{\mathcal{T}^*_{\mathsf{A}}}\ f\ (\mathbf{conode}\ k) &= \mathbf{conode}\ (\lambda b.\ \mathsf{map}_{\mathcal{T}^*_{\mathsf{A}}}\ (\mathbf{cons}\ b)\ (\mathsf{lang}_{\mathcal{T}^*_{\mathsf{A}}}\ (k\ b))).
\end{aligned}$$

We then define a continuous extension for filtering by a Boolean-valued predicate:

Definition 112 ($\mathsf{filter}_{\mathcal{T}^*_{\mathsf{A}}}$)**.** *For type A and predicate $P : A \to \mathbb{B}$, define* $\mathsf{filter}_{\mathcal{T}^*_{\mathsf{A}}}\ P : \mathcal{T}^*_A \to \mathcal{T}^*_A \triangleq (\mathsf{filter}_{\mathcal{T}_{\mathsf{A}}}\ P)^{\mathsf{co}}$*, where:*

$$\mathsf{filter}_{\mathcal{T}_{\mathsf{A}}}\ P \triangleq$$
$$\mathsf{fold}_{\mathcal{T}_{\mathsf{A}}}\ \bot_{\mathcal{T}^*_A}\ (\lambda a.\ \mathit{if}\ P\ a\ \ \mathit{then}\ \mathbf{coleaf}\ a\ \mathit{else}\ \bot_{\mathcal{T}^*_A})\ \mathbf{cotau}\ \mathbf{conode}$$

with computation rules:

$$\begin{aligned}
\mathsf{filter}_{\mathcal{T}^*_{\mathsf{A}}}\ P\ \bot_{\mathcal{T}^*_A} &= \bot_{\mathcal{T}^*_A} \\
\mathsf{filter}_{\mathcal{T}^*_{\mathsf{A}}}\ P\ (\mathbf{coleaf}\ a) &= \mathit{if}\ P\ a\ \mathit{then}\ \ \mathbf{coleaf}\ a\ \mathit{else}\ \bot_{\mathcal{T}^*_A} \\
\mathsf{filter}_{\mathcal{T}^*_{\mathsf{A}}}\ P\ (\mathbf{cotau}\ t) &= \mathbf{cotau}\ (\mathsf{filter}_{\mathcal{T}^*_{\mathsf{A}}}\ P\ t) \\
\mathsf{filter}_{\mathcal{T}^*_{\mathsf{A}}}\ P\ (\mathbf{conode}\ k) &= \mathbf{conode}\ (\mathsf{filter}_{\mathcal{T}^*_{\mathsf{A}}}\ P \circ k).
\end{aligned}$$

Notice that, in contrast to $\mathsf{filter}_{\mathcal{L}^*_{\mathsf{A}}}$ (Definition 81), elements removed by $\mathsf{filter}_{\mathcal{T}^*_{\mathsf{A}}}$ are simply replaced by $\bot_{\mathcal{T}^*_A}$ rather than restructuring the tree to eliminate them entirely. This is no problem because we never compute with preimage sets – they are only part of the correctness specification for the cotree equidistribution theorem (Theorem 23).

The preimage of a predicate Q with respect to sampler t is then obtained as the language of t after being filtered by Q.

Definition 113 ($\mathsf{preimage}_{\mathcal{T}^*_{\mathsf{A}}}$)**.** *For type A and predicate $Q : A \to \mathbb{B}$, define $\mathsf{preimage}_{\mathcal{L}^*_{\mathsf{A}}} : \mathcal{T}^*_A \to \mathcal{T}^*_{\mathcal{L}_{\mathbb{B}}}$ as:*

$$\mathsf{preimage}_{\mathcal{L}^*_{\mathsf{A}}} \triangleq \mathsf{lang}_{\mathcal{T}^*_{\mathsf{A}}} \circ \mathsf{filter}_{\mathcal{T}^*_{\mathsf{A}}}\, Q.$$

We define pairwise disjointness of cotree-encoded sets and prove that cotree preimages are always pairwise disjoint:

Definition 114 (Incomparable)**.** *For ordered type A, $x, y : A$ are incomparable (written $x \bowtie y$) when $\neg(x \sqsubseteq y \vee y \sqsubseteq x)$.*

Definition 115 ($\mathsf{disjoint}_{\mathcal{T}^*_{\mathsf{A}}}$)**.** *For ordered type A, define $\mathsf{disjoint}_{\mathcal{T}^*_{\mathsf{A}}} : \mathcal{T}^*_A \to \mathbb{P} \triangleq \mathsf{disjoint}_{\mathcal{T}_{\mathsf{A}}}{}^{\hat{\mathsf{co}}}$, where $\mathsf{disjoint}_{\mathcal{T}_{\mathsf{A}}} : \mathcal{T}_A \to \mathbb{P}$ is given inductively by the inference rules:*

DISJOINT-BOT
$$\frac{}{\mathsf{disjoint}_{\mathcal{T}_{\mathsf{A}}}\ \bot_{\mathcal{T}_A}}$$

DISJOINT-LEAF
$$\frac{a : A}{\mathsf{disjoint}_{\mathcal{T}_{\mathsf{A}}}\ (\mathbf{leaf}\ a)}$$

DISJOINT-TAU
$$\frac{\mathsf{disjoint}_{\mathcal{T}_{\mathsf{A}}}\ t}{\mathsf{disjoint}_{\mathcal{T}_{\mathsf{A}}}\ (\mathbf{tau}\ t)}$$

DISJOINT-NODE
$$\frac{\forall\, b : \mathbb{B},\ \mathsf{disjoint}\ (k\ b) \qquad \forall_{\lambda x.\ \forall_{x \bowtie}\ (k\ \mathbf{false})}\ (k\ \mathbf{true}) \qquad \forall_{\lambda x.\ \forall_{x \bowtie}\ (k\ \mathbf{true})}\ (k\ \mathbf{false})}{\mathsf{disjoint}_{\mathcal{T}_{\mathsf{A}}}\ (\mathbf{node}\ k)}$$

Theorem 22 (Cotree preimages are pairwise disjoint)**.** *Let A be a type, $P : A \to \mathbb{B}$, and $t : \mathcal{T}^*_A$. Then,*

$$\mathsf{disjoint}_{\mathcal{T}^*_{\mathsf{A}}}\ (\mathsf{preimage}_{\mathcal{T}^*_{\mathsf{A}}}\ P\ t).$$

We define the desired measure on Σ_1^0 sets as an application of the $\mathsf{sum}_{\mathcal{T}_A^*}$ operator for summing a function over a cotree.

Definition 116 ($\mathsf{sum}_{\mathcal{T}^*}$)**.** *For type A and $f : A \to \mathbb{R}_{\geq 0}^{\infty}$, define* $\mathsf{sum}_{\mathcal{T}_A^*}\ f : \mathcal{T}_A^* \to \mathbb{R}_{\geq 0}^{\infty} \triangleq \mathsf{sum}_{\mathcal{T}_A}\ f^{\mathsf{co}}$*, where:*

$$\mathsf{sum}_{\mathcal{T}_A} \triangleq \mathsf{fold}_{\mathcal{T}_A}\ 0\ f\ (\lambda g.\ g\ \mathbf{true} + g\ \mathbf{false})$$

with computation rules:

$$\mathsf{sum}_{\mathcal{T}_A^*}\ f\ (\mathbf{coleaf}\ a) = f\ x$$

$$\mathsf{sum}_{\mathcal{T}_A^*}\ f\ (\mathbf{conode}\ k) =$$

$$\mathsf{sum}_{\mathcal{T}_A^*}\ f\ (k\ \mathbf{true}) + \mathsf{sum}_{\mathcal{T}_A^*}\ f\ (k\ \mathbf{false}).$$

Definition 117 ($\mathsf{mu}_{\mathcal{T}_{\mathcal{L}_{\mathbb{B}}}^*}$)**.** *Define* $\mathsf{mu}_{\mathcal{T}_{\mathcal{L}_{\mathbb{B}}}^*} : \mathcal{T}_{\mathcal{L}_{\mathbb{B}}}^* \to \mathbb{R}_{\geq 0}^{\infty} \triangleq \mathsf{sum}_{\mathcal{T}_{\mathcal{L}_{\mathsf{bool}}}^*}\ (\lambda l.\ \frac{1}{2^{(\mathsf{length}_{\mathcal{L}_{\mathbb{B}}}\ l)}})$.

Proving Equidistribution With AlgCo In this section we use AlgCo to prove that that any sequence of samples produced by a cotree sampler t is *equidistributed* wrt. the $\mathsf{wp}_{\mathcal{T}_A^*}$ semantics of t. At a high level, our strategy is to "push forward" through the sampler the assumption of uniform distribution of the source of random bits. We first need the following two lemmas:

Lemma 23 ($\mathsf{wp}_{\mathcal{T}_A^*}\ \mathsf{filter}_{\mathcal{T}_A^*}$)**.** *Let A be a type and $Q : A \to \mathbb{B}$. Then,*

$$\mathsf{wp}_{\mathcal{T}_A^*}\ [Q] = \mathsf{wp}_{\mathcal{T}_A^*}\ \mathbf{1} \circ \mathsf{filter}_{\mathcal{T}_A^*}\ Q.$$

Lemma 24 ($\mathsf{wp}_{\mathcal{T}_A^*}\ \mathbf{1}$ equals $\mathsf{mu}_{\mathcal{T}_A^*} \circ \mathsf{lang}_{\mathcal{T}_A^*}$)**.** *Let A be a type and $Q : A \to \mathbb{B}$. Then,*

$$\mathsf{wp}_{\mathcal{T}_A^*}\ \mathbf{1} = \mathsf{mu}_{\mathcal{T}_A^*} \circ \mathsf{filter}_{\mathcal{T}_A^*}\ Q.$$

The proofs of Lemmas 23 and 24 both proceed by fusing the RHS to a single comorphism and applying Lemma 6 to reduce the goal to straightforward induction over finite trees. It immediately follows that the probability of any event Q according to the $\mathsf{wp}_{\mathcal{T}_A^*}$ semantics of sampler t coincides with the measure of the preimage of Q under f_t:

Lemma 25 ($\mathsf{wp}_{\mathcal{T}^*_A}$ is measure of preimage)**.** *Let A be a type, $Q : A \to \mathbb{B}$, and $t : \mathcal{T}^*_A$. Then,*

$$\mathsf{wp}_{\mathcal{T}^*_A}\,[Q]\,t = \mathsf{mu}_{\mathcal{T}^*_{\mathcal{L}_\mathbb{B}}}\,(\mathsf{preimage}_{\mathcal{T}^*_A}\,Q\,t).$$

To specify uniform distribution of the source of random bits, we use a variation of the classic notion of "uniform distribution modulo 1" [KN12, BG22] generalized to Σ^0_1 subsets of $2^\mathbb{N}$.

Definition 118 (Σ^0_1-u.d.)**.** *A sequence $\{x_i\}$ of bitstreams is Σ^0_1-uniformly distributed (Σ^0_1-u.d.) when for every $U : \Sigma^0_1$, $\lim_{n\to\infty} \frac{1}{n} \sum_{i=0}^{n} [\exists^{\mathsf{co}}_{=x_i}\, U] = \mathsf{mu}_{\mathcal{T}^*_{\mathcal{L}_\mathbb{B}}}\, U$.*

In other words, a sequence of bitstreams is uniformly distributed when every Σ^0_1 set gets its "proper share" of samples as the number of samples goes to infinity.

Theorem 23 (Equidistribution of samples)**.** *Let A be a type, $t : \mathcal{T}^*_A$, $Q : A \to \mathbb{B}$, $\{x_n\}$ a Σ^0_1-u.d. sequence of bitstreams, and $\{f_t(x_n)\}$ a sequence of samples obtained by mapping f_t over $\{x_n\}$. Then, $\{f_t(x_n)\}$ is $\mathsf{wp}_{\mathcal{T}^*_A}$-equidistributed wrt. t:*

$$\lim_{n\to\infty} \frac{1}{n} \sum_{i=0}^{n} [Q\,(f_t(x_i))] = \mathsf{wp}_{\mathcal{T}^*_A}\, t\, [Q]\, \sigma$$

Proof. Rewrite the RHS by Lemma 25 so the goal becomes:

$$\lim_{n\to\infty} \frac{1}{n} \sum_{i=0}^{n} [Q\,(f_t(x_i))] = \mathsf{mu}_{\mathcal{T}^*_{\mathcal{L}_\mathbb{B}}}\,(\mathsf{preimage}_{\mathcal{T}^*_A}\,Q\,t).$$

Then let $U = \mathsf{preimage}_{\mathcal{T}^*_A}\, Q\, t$ in the assumption of Σ^0_1-u.d. to obtain:

$$\lim_{n\to\infty} \frac{1}{n} \sum_{i=0}^{n} [\exists^{\mathsf{co}}_{=x_i}\, \mathsf{preimage}_{\mathcal{T}^*_A}\, Q\, t] = \mathsf{mu}_{\mathcal{T}^*_{\mathcal{L}_\mathbb{B}}}\, \mathsf{preimage}_{\mathcal{T}^*_A}\, Q\, t$$

from which the goal immediately follows since $Q\,(f_t(x_i)) \iff \exists^{\mathsf{co}}_{=x_i}\, \mathsf{preimage}_{\mathcal{T}^*_A}\, Q\, t$. □

7.5 Compiling CF Trees to Cotrees

In this section we show how CF trees are interpreted in Zar as cotrees in a manner that parallels the compilation from CF trees to interaction trees described in Section 4.5,

and prove correctness (Theorem 24) of the compilation with respect to weakest pre-expectation semantics of CF trees (Definition 40) and cotrees (Definition 110). Section 7.6 builds on the results of this section to prove semantics preservation of the interaction tree backend in Section 4.5, which ultimately leads to the ITree equidistribution theorem (Theorem 9) in Section 5.3.

7.5.1 Cotree Iteration Combinator

The main challenge in compiling CF trees to cotrees is in providing an iteration combinator with type $\forall\, I\, A,\ (I \to \mathcal{T}^*_{I+A}) \to I \to \mathcal{T}^*_A$ for compiling **fix** nodes, to match the use of ITree.iter in Definition 45. It can't be defined as a continuous extension because the type $I \to \mathcal{T}^*_{I+A}$ is not an algebraic CPO in general (and not in particular when I is specialized to the type Σ of program states). Thus we define $\mathsf{iter}_{\mathcal{T}^*}$ as follows:

Definition 119 ($\mathsf{iter}_{\mathcal{T}^*}$)**.** *For types I and A, $f : I \to \mathcal{T}^*_{I+A}$, and $z : I$, define $\mathsf{iter}_{\mathcal{T}^*_A}\ f\ z : \mathcal{T}^*_A \triangleq \mathsf{sup}\ (\mathsf{F}^n\ (\lambda\_.\ \mathbf{cobot}^{)})\ z$, where $\mathsf{F} : (I \to \mathcal{T}^*_A) \to I \to \mathcal{T}^*_A$ is given by:*

$$\begin{aligned}\mathsf{F}\ g\ i \triangleq f\ i \gg\!= \lambda lr.\ &\mathit{match}\ lr\ \mathit{with}\\ &\mid \mathsf{inl}\ j \Rightarrow \mathbf{cotau}\ (g\ j)\\ &\mid \mathsf{inr}\ x \Rightarrow \mathbf{coleaf}\ x\\ &\mathit{end}.\end{aligned}$$

```
cotree_iter =
  \ f i ->
    cotree_bind (f i) (\ lr ->
      case lr of
        Inl j -> cotree_iter f j
        Inr x -> coleaf x)
```

Figure 7.5: Haskell extraction primitive for $\mathsf{iter}_{\mathcal{T}^*}$.

Figure 7.5 shows a Haskell extraction primitive for $\mathsf{iter}_{\mathcal{T}^*}$ (not currently used by the Zar system but could in principle allow for a purely executable cotree backend (without the need for **cotau** nodes!)), justified by the following computation rule:

Lemma 26 ($\mathsf{iter}_{\mathcal{T}^*}$ computation rule)**.** *Let I and A be types, $f : I \to \mathcal{T}^*_{I+A}$, and $i : I$. Then,*

$$\begin{aligned}
&\mathsf{iter}_{\mathcal{T}^*_A}\ f\ i = f\ i \gg\!= \lambda lr.\ \mathit{match}\ lr\ \mathit{with}\\
&\qquad\qquad \mid \mathsf{inl}\ j \Rightarrow \mathbf{cotau}\ (\mathsf{iter}_{\mathcal{T}^*_A}\ f\ j)\\
&\qquad\qquad \mid \mathsf{inr}\ x \Rightarrow \mathbf{coleaf}\ x\\
&\qquad\qquad \mathit{end}.
\end{aligned}$$

With $\mathsf{iter}_{\mathcal{T}^*}$, we can compile CF trees to cotrees of type $\mathcal{T}^*_{\mathbf{1}+\Sigma}$, where $\mathbf{1}$ in the LHS of the sum type encodes observation failure:

Definition 120 (to_cotree_open)**.** *Given an unbiased CF tree $t : \mathcal{T}^{cf}_\Sigma$, define* to_cotree_open $t : \mathcal{T}^*_{\mathbf{1}+\Sigma}$ *by induction on t:*

$$\begin{aligned}
&\mathsf{to_cotree_open} : \mathcal{T}^{cf}_\Sigma \to \mathcal{T}^*_{\mathbf{1}+\Sigma}\\
&\mathbf{leaf}\ \sigma \triangleq \mathbf{coleaf}\ (\mathsf{inr}\ \sigma)\\
&\mathbf{fail} \triangleq \mathbf{coleaf}\ (\mathsf{inl}\ ())\\
&\mathbf{choice}\ _\ k \triangleq \mathbf{conode}\ (\mathsf{to_cotree_open} \circ k)\\
&\mathbf{fix}\ \sigma_0\ e\ g\ h \triangleq \mathsf{iter}_{\mathcal{T}^*_\Sigma}\ (\lambda\sigma.\ \mathit{if}\ e\ \sigma\ \mathit{then}\\
&\qquad\qquad \mathsf{to_cotree_open}\ (\mathsf{g}\ \mathsf{y}) \gg\!=\\
&\qquad\qquad\ \lambda\, lr.\ \mathit{match}\ lr\ \mathit{with}\\
&\qquad\qquad\qquad \mid \mathsf{inl}\ _ \Rightarrow \mathbf{coleaf}\ (\mathsf{inr}\ (\mathsf{inl}\ ()))\\
&\qquad\qquad\qquad \mid \mathsf{inr}\ z \Rightarrow \mathbf{coleaf}\ (\mathsf{inl}\ z)\\
&\qquad\qquad\qquad \mathit{end}\\
&\qquad\qquad \mathit{else}\\
&\qquad\qquad\ \mathsf{map}_{\mathcal{T}^*_\Sigma}\ \mathsf{inr}\ ())\ \sigma_0
\end{aligned}$$

We then "tie the knot" analogously to Definition 46 to transform cotrees of type $\mathcal{T}^*_{\mathbf{1}+\Sigma}$ ("open" cotrees) into ones of type $\mathcal{T}^*_\Sigma$ ("closed") where all occurrences of inl () are replaced by recurrences to the root:

Definition 121 (tie_cotree)**.** *For* $t : \mathcal{T}^*_{\mathbf{1}+\Sigma}$*, define* tie_cotree $t : \mathcal{T}^{it}_\Sigma$ *as:*

$$\mathsf{tie\_cotree}\ t \triangleq \mathsf{iter}_{\mathcal{T}^*_\Sigma}\ (\lambda\_.\ t)\ (),$$

and the overall compilation from CF trees to cotrees is given by the composition of tie_cotree after to_cotree_open:

Definition 122 (to_cotree)**.** *For* $t : \mathcal{T}^{cf}_\Sigma$*, define* to_cotree $t : \mathcal{T}^{it}_\Sigma$ *as:*

$$\mathsf{to\_cotree}\ t \triangleq \mathsf{tie\_cotree}\ (\mathsf{to\_cotree\_open}\ t).$$

We then prove that to_cotree_open preserves wp, wlp, and cwp semantics of CF trees via Lemmas 27, 28, and 29, respectively:

Lemma 27 (to_cotree_open $\mathsf{wp}_{\mathcal{T}^*_\Sigma}$)**.** *Let* $b : \mathbb{B}$*,* $t : \mathcal{T}^{cf}_\Sigma$ *an unbiased CF tree, and* $f : \Sigma \to \mathbb{R}^\infty_{\geq 0}$*. Then,*

$$\mathsf{twp_b}\ t\ f = \mathsf{wp}_{\mathcal{T}^*_\Sigma}\ ([\mathsf{b}] + f)\ (\mathsf{to\_cotree\_open}\ t).$$

Lemma 28 (to_cotree_open $\mathsf{wlp}_{\mathcal{T}^*_\Sigma}$)**.** *Let* $b : \mathbb{B}$*,* $t : \mathcal{T}^{cf}_\Sigma$ *an unbiased CF tree, and* $f : \Sigma \to \mathbb{R}^{\leq 1}_{\geq 0}$*. Then,*

$$\mathsf{twlp_b}\ t\ f = \mathsf{wlp}_{\mathcal{T}^*_\Sigma}\ ([\mathsf{b}] + f)\ (\mathsf{to\_cotree\_open}\ t).$$

Lemma 29 (to_cotree_open $\mathsf{cwp}_{\mathcal{T}^*_\Sigma}$)**.** *Let* $b : \mathbb{B}$*,* $t : \mathcal{T}^{cf}_\Sigma$ *an unbiased CF tree, and* $f : \Sigma \to \mathbb{R}^\infty_{\geq 0}$*. Then,*

$$\mathsf{tcwp}\ t\ f = \mathsf{cwp}_{\mathcal{T}^*_\Sigma}\ f\ (\mathsf{to\_cotree\_open}\ t).$$

As tie_cotree replaces observation failures with recurrences to the root of the tree, cwp reasoning on an "open" $t : \mathcal{T}^*_{\mathbf{1}+\Sigma}$ can be replaced with plain-old wp reasoning on its "closed" form tie_cotree $t : \mathcal{T}^*_\Sigma$:

Lemma 30 (tie_cotree $\mathsf{cwp}_{\mathcal{T}^*_\Sigma}$)**.** *Let A be a type, $t : \mathcal{T}^*_{\mathbf{1}+A}$, and $f : A \to \mathbb{R}^\infty_{\geq 0}$ such that* $\mathsf{wp}_{\mathcal{T}^*_A}\ (\mathbf{1} + \mathbf{0})\ t < 1$. *Then,*

$$\mathsf{cwp}_{\mathcal{T}^*_A}\ f\ t = \mathsf{wp}_{\mathcal{T}^*_A}\ f\ (\mathsf{tie\_cotree}\ t).$$

The precondition $\mathsf{wp}_{\mathcal{T}^*_A}\ (\mathbf{1} + \mathbf{0})\ t < 1$ in the above lemma asserts that the probability of observation failure in t is less than 1, i.e., that the program it was compiled from did not condition on contradictory observations. Lemmas 29 and 30 together imply the overall correctness of cotree compilation (where precondition $\mathsf{twp}_{\mathbf{true}}\ t\ \mathbf{0} < 1$ again ensures consistency of conditioned observations):

Theorem 24 (Correctness of cotree compilation)**.** *Let $t : \mathcal{T}^{cf}_\Sigma$ be an unbiased CF tree and $f : \Sigma \to \mathbb{R}^\infty_{\geq 0}$ such that* $\mathsf{twp}_{\mathbf{true}}\ t\ \mathbf{0} < 1$. *Then,*

$$\mathsf{wp}_{\mathcal{T}^*_\Sigma}\ f\ (\mathsf{to\_cotree}\ t) = \mathsf{tcwp}\ t\ f.$$

The following section defines a weakest pre-expectation style semantics on interaction trees in terms of $\mathsf{wp}_{\mathcal{T}^*}$ and extends the result of Theorem 24 to apply to the interaction tree backend from Section 4.5.

7.6 Relating to Interaction Trees

Our strategy for verifying the correctness of the interaction tree backend of Section 4.5 (i.e., for proving Theorem 8) is to extend the result of Theorem 24 by:

1. Defining a wp semantics on ITrees via $\mathsf{wp}_{\mathcal{T}^*}$ through an injective mapping icotree : $\forall\ A.\ \mathcal{T}^{it}_A \to \mathcal{T}^*_A$ from ITrees to cotrees,

2. proving that ITrees and cotrees that are congruent in structure (Definition 125) are semantically equivalent with respect to their respective wp semantics, and

3. proving that the ITrees and cotrees generated by to_itree (Definition 47) and to_cotree (Definition 122) are indeed congruent in structure and thus semantically equivalent.

We begin with the injection of ITrees into the type of cotrees:

Definition 123 (icotree)**.** *For $t : \mathcal{T}_A^{it}$ an ITree sampler with element type A, define* icotree $t : \mathcal{T}_A^{*}$ *by coinduction:*

$$\begin{array}{|l|}\hline \mathsf{icotree} : \mathcal{T}_A^{it} \to \mathcal{T}_A^{*} \\ \hline \end{array}$$

$$\begin{aligned} \mathsf{icotree}\ (\mathbf{ret}\ a) &\triangleq \mathbf{coleaf}\ a \\ \mathsf{icotree}\ (\mathbf{tau}\ t) &\triangleq \mathbf{cotau}\ (\mathsf{icotree}\ t) \\ \mathsf{icotree}\ (\mathbf{vis}\ \mathbf{get}\ k) &\triangleq \mathbf{conode}\ (\mathsf{icotree} \circ k) \end{aligned}$$

At this point the reason for having **cotau** nodes in the type of cotrees becomes clear, for without them we would be unable to define icotree via primitive recursion, making the results of this section more difficult to obtain (e.g., attempting to describe the mapping via a coinductive relation would be troubled by issues described in Section 6.1, namely the need to explicitly rule out degenerate cases involving infinitely nested **tau**s).

With icotree can now easily define a wp semantics on ITrees by reduction to $\mathsf{wp}_{\mathcal{T}^*}$:

Definition 124 ($\mathsf{wp}_{\mathcal{T}_A^{it}}$)**.** *For type A, $f : A \to \mathbb{R}_{\geq 0}^{\infty}$, and $t : \mathcal{T}_A^{it}$, define $\mathsf{wp}_{\mathcal{T}_A^{it}}\ f\ t : \mathbb{R}_{\geq 0}^{\infty}$ as:*

$$\mathsf{wp}_{\mathcal{T}_A^{it}}\ f\ t \triangleq \mathsf{wp}_{\mathcal{T}_A^{*}}\ f\ (\mathsf{icotree}\ t).$$

Next we define a two-place relation between ITrees and cotrees that holds whenever they are structurally congruent:

Definition 125 ($\equiv_{\mathcal{T}}$)**.** *For type A, define $\equiv_{\mathcal{T}_A} : \mathcal{T}_A^{it} \to \mathcal{T}_A^{*} \to \mathbb{P}$ coinductively by the inference rules:*

$\equiv_{\mathcal{T}}$-RET

$$\frac{a : A}{\textbf{ret}\ a \equiv_{\mathcal{T}_A} \textbf{coleaf}\ a}$$

$\equiv_{\mathcal{T}}$-TAU

$$\frac{t_1 \equiv_{\mathcal{T}_A} t_2}{\textbf{tau}\ t_1 \equiv_{\mathcal{T}_A} \textbf{cotau}\ t_2}$$

$\equiv_{\mathcal{T}}$-NODE

$$\frac{\forall b : \mathbb{B},\ f\ b \equiv_{\mathcal{T}_A} g\ b}{\textbf{vis get}\ f \equiv_{\mathcal{T}_A} \textbf{conode}\ g}$$

and show that $\equiv_{\mathcal{T}}$ implies semantic equivalence:

Lemma 31 ($\equiv_{\mathcal{T}}$ implies semantic equivalence)**.** *Let A be a type, $it : \mathcal{T}_A^{it}$, $ct : \mathcal{T}_A^{*}$, and $f : A \to \mathbb{R}^{\infty}_{\geq 0}$ such that $it \equiv_{\mathcal{T}_A} ct$. Then,*

$$\mathsf{wp}_{\mathcal{T}_A^{it}}\ f\ it = \mathsf{wp}_{\mathcal{T}_A^{*}}\ f\ ct.$$

Our goal is now reduced to proving that to_itree (Definition 47) and to_cotree (Definition 122) produce congruent structures, from which semantic equivalence immediately follows. To that end, we provide a series of lemmas for proving $\equiv_{\mathcal{T}}$ for map, bind, and iter constructions:

Lemma 32 ($\equiv_{\mathcal{T}}$ map)**.** *Let A and B be types, $f : A \to B$, $it : \mathcal{T}_A^{it}$, and $ct : \mathcal{T}_A^{*}$ such that $it \equiv_{\mathcal{T}_A} ct$. Then,*

$$\mathsf{ITree.map}\ f\ it \equiv_{\mathcal{T}_B} \mathsf{map}_{\mathcal{T}_A^{*}}\ f\ ct.$$

Lemma 33 ($\equiv_{\mathcal{T}}$ bind)**.** *Let A and B be types, $it : \mathcal{T}_A^{it}$, $ct : \mathcal{T}_A^{*}$, $f : A \to \mathcal{T}_B^{it}$, and $g : A \to \mathcal{T}_B^{*}$ such that $it \equiv_{\mathcal{T}_A} ct$ and $\forall b : \mathbb{B},\ f\ b \equiv_{\mathcal{T}_B} g\ b$. Then,*

$$\mathsf{ITree.bind}\ it\ f \equiv_{\mathcal{T}_B} ct \gg\!= g.$$

Lemma 34 ($\equiv_{\mathcal{T}}$ iter)**.** *Let I and A be types, $z : I$, $f : I \to \mathcal{T}_{I+A}^{it}$, and $g : I \to \mathcal{T}_{I+A}^{*}$ such that $\forall i : I,\ f\ i \equiv_{\mathcal{T}_{I+A}} g\ i$. Then,*

$$\mathsf{ITree.iter}\ f\ z \equiv_{\mathcal{T}_A} \mathsf{iter}_{\mathcal{T}_A^{*}}\ g\ z.$$

We use the above lemmas to prove congruence of to_itree_open (Definition 45) and to_cotree_open (Definition 120):

Lemma 35 (to_itree_open equivalent to to_cotree_open)**.** *Let* $t : \mathcal{T}_\Sigma^{cf}$ *be a CF tree. Then,*

$$\mathsf{to\_itree\_open}\ t \equiv_{\mathcal{T}_{\mathbf{1}+\Sigma}} \mathsf{to\_cotree\_open}\ t.$$

and finally congruence of outputs of to_itree and to_cotree:

Theorem 25 (to_itree equivalent to to_cotree)**.** *Let* $t : \mathcal{T}_\Sigma^{cf}$ *be a CF tree. Then,*

$$\mathsf{to\_itree}\ t \equiv_{\mathcal{T}_\Sigma} \mathsf{to\_cotree}\ t.$$

from which it immediately follows (in conjunction with Lemma 31) that $\mathsf{wp}_{\mathcal{T}_\Sigma^{it}}\ f\ (\mathsf{to\_itree}\ t) = \mathsf{wp}_{\mathcal{T}_\Sigma^{*}}\ f\ (\mathsf{to\_cotree}\ t)$ for all $t : \mathcal{T}_\Sigma^{cf}$ and $f : \Sigma \to \mathbb{R}^\infty_{\geq 0}$, and then (from Theorem 24) finally the correctness of ITree compilation:

Theorem 26 (Correctness of ITree compilation)**.** *Let* $t : \mathcal{T}_\Sigma^{cf}$ *be an unbiased CF tree and* $f : \Sigma \to \mathbb{R}^\infty_{\geq 0}$ *such that* $\mathsf{twp}_{\mathbf{true}}\ t\ \mathbf{0} < 1$*. Then,*

$$\mathsf{wp}_{\mathcal{T}_\Sigma^{it}}\ f\ (\mathsf{to\_itree}\ t) = \mathsf{tcwp}\ t\ f.$$

which when taken together with Theorem 3 implies the overall end-to-end compiler correctness result of Theorem 8.

8 Empirical Validation

> ... who knows whether proof of the
>
> devil is also a proof of God?
>
> *Ivan Fyodorovich*

This chapter provides empirical validation of the following aspects of samplers compiled from cpGCL programs:

• *Correctness.* To validate Theorem 10, we compare the empirical distribution of generated samples with the expected true distribution with respect to total variation (TV) distance, Kullback-Leibler (KL) divergence [KL51], and SMAPE[1] [Arm85].

• *Performance.* Although generated samplers are not guaranteed to be entropy-optimal (in contrast to OPTAS [SFRM20b]), we measure statistics of the number of uniform random bits required to obtain a sample.

We do not verify the programs in the following sections with respect to their cwp semantics (except for Figure 1.1a), as we seek only to validate the correctness of the compilation pipeline.

OCaml Shim All programs in this chapter are compiled to verified ITree samplers (as described in Section 4.5) and extracted to OCaml [Let08, LDF+21] for execution by the driver code in Figure 8.1. Thus, correctness of extracted samplers depends on the PRNG provided by the OCaml Random module being Σ_1^0-u.d. (Def. 118). Sample records are generated and written to disk for external analysis with handwritten Python code (see, e.g., /extract/die/analyze.py) and statistics routines provided by scipy.stats [com22])

Trusted Computing Base Our TCB includes the Coq typechecker (and therefore the OCaml compiler and runtime), the specifications of cwp (Section 3.5) and equidistribution (Section 5.3), and the OCaml extraction mechanism and driver code in Figure 8.1.

[1] Symmetric Mean Absolute Percentage Error

```
let _ = Random.self_init ()  (* Seed PRNG. *)
let rec run t =              (* t : (boolE, 'a) itree *)
  match observe t with       (* Unfold ITree. *)
  | RetF x -> x              (* Produce sample. *)
  | TauF t' -> run t'        (* Skip tau node. *)
  | VisF (_, k) ->           (* Consume random bit. *)
      run (k (Obj.magic (Random.bool ())))
```

Figure 8.1: OCaml shim for execution of ITree samplers. The destructor 'observe' (not to be confused with the cpGCL command of the same name) unfolds the structure of ITree t.

Empirical Evaluation The remainder of this chapter contains tables showing results of empirical evaluation of accuracy and entropy-performance of a collection of illustrative cpGCL programs (all with respect to a sample size of 100,000). In each table, the first column denotes the values taken by the parameter of the distribution (e.g., the bias parameter p for Bernoulli, range n for uniform, etc.). μ_x and σ_x denote the mean and standard deviation of the posterior over variable x. TV, KL, and SMAPE denote the total variation distance, Kullback-Leibler (KL) divergence [KL51], and symmetric mean absolute percentage error [Arm85], respectively, of the empirical distribution with respect to the true distribution. μ_{bit} and σ_{bit} denote the mean and standard deviation of the number of uniform random bits required to obtain a sample.

Entropy Usage The Shannon entropy [Sha48] (or *information entropy*) of a probability distribution provides a lower bound on the average number of uniformly random bits required to obtain a single i.i.d. sample. Knuth and Yao [KY76] show that an "entropy-optimal" sampler in the random bit model consumes no more than 2 bits on average than the entropy of the encoded distribution. Our samplers are not guaranteed to be entropy optimal, but that is a possible direction for future work.

```
duel (p : ℚ) :=
  a ← false; b ← false;
  while a = b do flip a p; flip b p; end
```

(a) Dueling coins program with bias $p \in (0, 1)$.

```
die (n : ℕ) :=
  uniform n (λm. x ← m + 1)
```

(b) Rolling an n-sided die.

Figure 8.2: Dueling coins (left) and n-sided die (right) cpGCL programs.

Flip The command flip x p : cpGCL performs a probabilistic choice ("flips a coin") with probability $p : \mathbb{Q}$ of **true** (or "heads") and assigns the result to variable x.

Definition 126 (flip)**.** *For* x : ident *and* $p \in [0, 1] \subseteq \mathbb{Q}$*, define* flip x p : cpGCL *as:*

$$\text{flip } x\ p \triangleq \{\ x \leftarrow \textbf{true}\ \}\ [p]\ \{\ x \leftarrow \textbf{false}\ \}.$$

8.1 Dueling Coins

The *dueling coins* program (Figure 8.2a) illustrates an i.i.d. loop (unbounded but with no loop-carried dependence) simulating a fair coin using a biased one. The posterior distribution over a is Bernoulli($\frac{1}{2}$) for any input bias $p \in (0, 1)$. The dueling coins illustrate a situation in which the average number of bits required to obtain a sample ($\mu_{bit} \sim 12$ when $p = \frac{2}{3}$ and $\mu_{bit} \sim 135$ when $p = \frac{1}{20}$, see Figure 8.1) substantially exceeds the entropy (exactly 1) of the posterior.

8.2 Geometric Primes

The *geometric primes* program (Figure 1.1a) illustrates the use of a *non-i.i.d.* loop and conditioning as follows: Repeatedly flip a coin with bias p, counting the number of heads until landing one tails. Finally, observe that the number of heads counted is a prime number. What, then, is the posterior over the number of heads h? The true posterior over prime h is given by the probability mass function (pmf):

p	μ_a	σ_a	TV	KL	SMAPE	μ_{bit}	σ_{bit}
2/3	0.50	0.50	2.02×10^{-3}	1.20×10^{-5}	2.02×10^{-3}	12.0	9.39
4/5	0.50	0.50	2.16×10^{-3}	1.30×10^{-5}	2.16×10^{-3}	27.59	23.49
1/20	0.50	0.50	2.83×10^{-3}	2.30×10^{-5}	2.83×10^{-3}	134.97	129.07

Table 8.1: Accuracy and entropy usage for Prog. 8.2a with $p = \frac{2}{3}$, $\frac{4}{5}$, and $\frac{1}{20}$. mu_{bit} and σ_{bit} increase as p is goes further from $\frac{1}{2}$ due to increasing Shannon entropy of Bernoulli(p).

p	μ_h	σ_h	TV	KL	SMAPE	μ_{bit}	σ_{bit}
1/2	2.64	1.10	2.33×10^{-3}	6.40×10^{-5}	7.63×10^{-2}	9.66	7.21
2/3	3.24	1.93	2.48×10^{-3}	1.10×10^{-4}	4.12×10^{-2}	25.31	20.59
1/5	2.19	0.44	7.44×10^{-4}	5.0×10^{-6}	5.19×10^{-3}	142.51	132.70

Table 8.2: Accuracy and entropy usage for Prog. 1.1a with $p = \frac{1}{2}$, $\frac{2}{3}$, and $\frac{1}{5}$. μ_{bit} and σ_{bit} are high when $p = \frac{1}{5}$ due to low probability of 'h is prime', illustrating a general weakness (entropy waste) of our rejection samplers when conditioning on low-probability events.

$\mathsf{Pr}(X = h \mid h \text{ is prime}) = \frac{(1-p)^{h+1}}{\sum_{k \in \mathcal{P}} (1-p)^{k+1}}$, where $\mathcal{P}$ denotes the set of prime numbers. Figure 8.2 shows accuracy and entropy statistics of the corresponding sampler compiled by Zar.

8.3 Uniform Sampling

Figure 8.3 shows accuracy and entropy usage for Prog. 8.2b for $n = 6$, 200, and 10000.

Zar and TensorFlow 2 We provide a Python 3 package (built from extracted samplers using pythonlib [Cap22]) exposing a simple interface for constructing and generating samples from verified uniform samplers. To demonstrate the use of Zarpy as a high-assurance drop-in replacement for unverified samplers, we implement a TensorFlow

n	μ_h	σ_h	TV	KL	SMAPE	μ_{bit}	σ_{bit}
6	3.49	1.71	3.86×10^{-3}	5.80×10^{-5}	3.87×10^{-3}	3.66	1.33
200	100.42	57.65	1.77×10^{-2}	1.36×10^{-3}	1.77×10^{-2}	9.01	2.18
10k	5011.87	2892.0	1.24×10^{-1}	7.33×10^{-2}	1.28×10^{-1}	15.62	2.74

Table 8.3: Accuracy and entropy usage for Prog. 8.2b with $n = 6$, 200, and 10k (with Shannon entropies 2.59, 7.64, and 13.29, respectively). μ_{bit} and σ_{bit} therefore show relatively good performance with near entropy-optimality.

2 [RM19] project (/python/tf/ in the source directory) for training an MNIST [LBBH98] classifier via stochastic gradient descent. We observe a negligible effect on training performance and excellent accuracy on the test set, as expected.

Comparison with FLDR and OPTAS The Fast Loaded Dice Roller (FLDR) [SFRM20a] is a time- and space-efficient algorithm for rolling an n-sided die, with implementations available in Python and C. Related to FLDR is OPTAS [SFRM20b], a system for optimal approximate sampling from discrete distributions with respect to a user-specified number of random bits, also with implementations in Python and C. Figure 8.4 shows a comparison of a 200-sided die in FLDR and OPTAS (with 32-bit precision and the "hellinger" kernel) with OCaml and Python variants of Zar. Initialization time is negligible for both Zar and FLDR. The Python Zar wrapper (created via pythonlib [Cap22]) adds a slowdown on the order of 10x.

8.4 Discrete Gaussian

We define discrete variants of Laplace and Gaussian distributions (based on [CKS20]) as reusable subroutines for larger cpGCL programs (e.g., the hare and tortoise program in Section 8.5). These subroutines differ from flip in Def. 126 by making use of local variables. Although cpGCL lacks built-in support for procedure calls (which

	μ_x	σ_x	TV	KL	SMAPE	μ_{bit}	σ_{bit}	T_{init}	T_s
Zar (OCaml)	99.52	57.58	1.76×10^{-2}	1.53×10^{-3}	2.25×10^{-2}	8.99	2.16	<1ms	132ms
Zar (Py)	99.51	57.69	1.96×10^{-2}	1.82×10^{-3}	2.21×10^{-2}	9.0	2.18	<1ms	1.67s
FLDR (C)	99.39	57.79	1.96×10^{-2}	1.18×10^{-3}	2.21×10^{-2}	9.01	2.18	<1ms	16ms
FLDR (Py)	99.32	57.70	2.08×10^{-2}	1.36×10^{-3}	2.33×10^{-2}	9.0	2.16	<1ms	290ms
OPTAS (C)	99.50	57.74	1.85×10^{-2}	1.20×10^{-3}	2.10×10^{-2}	8.55	1.27	3ms	45ms
OPTAS (Py)	99.58	57.69	2.12×10^{-2}	1.37×10^{-3}	2.37×10^{-2}	8.55	1.27	15ms	330ms

Table 8.4: Comparison of 200-sided die samplers with output x. T_{init} denotes time elapsed over construction and initializion of the sampler, and T_s the total time to generate 100k samples.

can be done in principle, as in [OKKM16]), they can be shallowly embedded if we take careful account of variables modified (i.e, "clobbered") within subroutines.

8.4.1 Inverse Exponential Bernoulli

To sample from a discrete Laplace, we first require a subroutine for sampling from a Bernoulli distribution with inverse exponential bias. We begin with a preliminary routine (bernoulli_exponential_0_1) for the special case of $\mathbf{0} \leq \gamma \leq \mathbf{1}$, which modifies variables k and a (used as a counter and loop flag, respectively), followed by its generalization (bernoulli_exponential) to $\mathbf{0} \leq \gamma$, additionally modifying variables i and b (also a counter and loop flag).

```
bernoulli_exponential_0_1 (out : ident) (γ : Σ → ℚ) :=
  k ← 0; a ← true;
  while a do { k ← k + 1 } [γ/(k+1)] { a ← false} end;
  if even k then out ← true else out ← false end
```

Figure 8.3: Sampling from Bernoulli($\exp(-\gamma)$), where $\mathbf{0} \le \gamma \le \mathbf{1}$

```
bernoulli_exponential (out : ident) (γ : Σ → ℚ) :=
  if γ ≤ 1
  then bernoulli_exponential_0_1 out γ
  else i ← 1; b ← true;
       while b ∧ i ≤ γ do bernoulli_exponential_0_1 b 1; i ← i + 1 end;
       if b then bernoulli_exponential_0_1 out (γ − ⌊γ⌋) else out ← 0 end
```

Figure 8.4: Sampling from Bernoulli($\exp(-\gamma)$), where $\mathbf{0} \le \gamma$

γ	μ_{out}	σ_{out}	TV	KL	SMAPE	μ_{bit}	σ_{bit}
$1/2$	0.61	0.49	1.86×10^{-3}	1.0×10^{-5}	1.95×10^{-3}	2.54	2.16
$3/2$	0.23	0.42	1.36×10^{-3}	8.0×10^{-6}	1.96×10^{-3}	3.84	3.59
10	9.0×10^{-5}	9.49×10^{-3}	4.50×10^{-5}	2.50×10^{-5}	1.65×10^{-1}	4.56	5.11

Table 8.5: Accuracy and entropy usage for Figure 8.4.

8.4.2 Discrete Laplace

A discrete analogue $\mathsf{Lap}_{\mathbb{Z}}(b)$ [CKS20] of the Laplace distribution (useful for, e.g., differential privacy [GRS09], and as a subroutine for the discrete Gaussian in the following section) with scale parameter b is defined by the probability mass function

$\mathsf{Pr}_{\mathsf{Lap}_{\mathbb{Z}}(b)}(X = x) = \frac{e^{1/b}-1}{e^{1/b}+1} \cdot e^{-|x|/b}$. Figure 8.5 shows a cpGCL program for sampling from $\mathsf{Lap}_{\mathbb{Z}}(\frac{t}{s})$ for positive integers s and t.

```
laplace (out : ident) (s t : ℕ) :=
  lp ← true;
  while lp do
    uniform t (λu.
      bernoulli_exponential d (λs. u/t);
      if d then
        v ← 0; bernoulli_exponential il 1;
        while il do v ← v + 1; bernoulli_exponential il 1 end;
          x ← u + t · v; y ← x/s; flip c 1/2;
          if c ∧ y = 0 then skip
          else lp ← false; out ← (1 − 2 · [c]) · y
      else skip)
  end
```

Figure 8.5: Sampling from $\mathsf{Lap}_{\mathbb{Z}}$. Modified variables: k, a, i, b, lp, d, v, il, x, y, and c. Variables lp and il ("loop" and "inner loop") are used for control flow. See [CKS20] for explanation and proof-of-correctness of the sampling algorithm.

s, t	μ_{out}	σ_{out}	TV	KL	SMAPE	μ_{bit}	σ_{bit}
$1, 2$	1.79×10^{-2}	2.81	3.51×10^{-3}	4.20×10^{-4}	1.64×10^{-1}	10.47	7.04
$2, 1$	1.79×10^{-3}	0.60	1.47×10^{-3}	7.10×10^{-5}	5.30×10^{-2}	9.77	8.17
$5, 2$	-8.50×10^{-4}	0.44	1.24×10^{-3}	1.09×10^{-4}	1.37×10^{-1}	15.53	12.38

Table 8.6: Accuracy and entropy usage for Figure 8.5 with scale parameter $\frac{t}{s}$.

8.4.3 Discrete Gaussian

A discrete analogue $\mathcal{N}_{\mathbb{Z}}(\mu, \sigma^2)$ [CKS20] of the Gaussian ("normal") distribution (useful for, e.g., lattice-based cryptography [ZSS20], and as a subroutine for the hare-and-tortoise program in Section 8.5) with parameters μ and σ is defined by the probability mass function $\mathsf{Pr}_{\mathcal{N}_{\mathbb{Z}}(\mu,\sigma^2)}(X = x) = \frac{e^{-(x-\mu)^2/2\sigma^2}}{\sum_{y:\mathbb{Z}} e^{-(y-\mu)^2/2\sigma^2}}$. Figure 8.6 shows a cpGCL program for sampling from $\mathcal{N}_{\mathbb{Z}}(\mu, \sigma^2)$.

```
gaussian_0 (z : ident) (σ : ℚ) :=
  ol ← false;
  while ¬ol do laplace z 1 ⌊σ⌋ + 1; bernoulli_exponential ol (λs. ((|z| − σ²/t)²)/(2σ²)) end

gaussian (out : ident) (μ : Σ → ℤ) (σ : ℚ) :=
  gaussian out σ; out ← out + μ
```

Figure 8.6: Sampling from $\mathcal{N}_{\mathbb{Z}}(\mu, \sigma^2)$. Note that the entropy usage depends only on σ and not μ. Modified variables: k, a, i, b, lp, d, v, il, x, y, c, ol, z. Variable ol ("outer loop") is used for control flow. See [CKS20] for explanation and proof-of-correctness of the sampling algorithm.

μ, σ	μ_z	σ_z	TV	KL	SMAPE	μ_{bit}	σ_{bit}
$0, 1$	-3.03×10^{-3}	1.0	2.71×10^{-3}	1.03×10^{-4}	4.49×10^{-2}	26.68	24.43
$10, 2$	10.0	2.0	3.69×10^{-3}	1.16×10^{-4}	7.22×10^{-2}	37.61	29.10
$-50, 5$	-50.01	5.01	6.11×10^{-3}	4.46×10^{-4}	5.70×10^{-2}	43.66	31.20

Table 8.7: Accuracy and entropy usage for Figure 8.6 with mean μ and variance σ^2.

```
hare_tortoise (P : Σ → ℙ) :=
  uniform 10 (λn. t_0 ← n);
  tortoise ← t_0; hare ← 0; time ← 0;
  while hare < tortoise do
    time ← time + 1;
    tortoise → tortoise + 1;
    { gaussian jump 4 2;
      hare ← hare + jump } [2/5] { skip }
  end;
  observe P
```

(a) cpGCL program simulating a race between a hare and tortoise.

P	μ_{t_0}	σ_{t_0}	μ_{bit}	σ_{bit}
true	4.49	2.87	193.88	220.06
$time \leq 10$	3.80	2.79	273.87	378.82
$time \geq 10$	6.18	2.31	596.68	359.85
$time \geq 20$	6.40	2.25	1376.74	930.20

(b) Accuracy and entropy usage for Figure 8.7a. μ_{t_0} and σ_{t_0} denote the mean and std deviation of the posterior over the tortoise's head start t_0, conditioned on P.

Figure 8.7: Hare and tortoise cpGCL program (left) with accuracy and entropy statistics (right).

8.5 Hare and Tortoise

Our final example shown in Figure 8.7a illustrates the use of the discrete Gaussian subroutine along with a non-i.i.d. loop and conditioning to simulate a race between a hare and a tortoise along a one-dimensional line, and the use of Zar to perform Bayesian inference [BT11]. The tortoise begins with uniformly-distributed head start t_0 and proceeds at a steady pace of 1 unit per time step. The hare begins at position 0 and occasionally (with probability $\frac{2}{5}$) leaps forward a Gaussian-distributed distance. The race ends when the hare reaches the tortoise, and then the terminal state is conditioned on predicate P. For example, by setting $P(time) = time \geq 10$ and querying the posterior over t_0, we ask: "Given that it took at least 10 time steps for the hare to reach the tortoise, what are likely values for the tortoise's head start?" (see Figure 8.7b).

9 Related Work

In this chapter we discuss related work not covered in previous chapters (see Section 8.3 for comparison with FLDR [SFRM20a] and OPTAS [SFRM20b] samplers. Works related generally to the Zar system appear in Section 9.1, and to the particular framework of Algebraic Coinductives in Section 9.2.

9.1 Zar

Compilation of Probabilistic Programs Holtzen et al. [HMB19, HdBM20] compile discrete probabilistic programs with bounded loops and conditioning to a symbolic representation based on binary decision diagrams (BDDs) [DM02, Ake78], exploiting independence of variables for efficient exact inference. Our CF trees are not as highly optimized (and we currently do not support exact inference), but we remark that BDDs, representing *finite* Boolean functions, are fundamentally insufficient for programs with unbounded loops for which no upper bound can be placed on the number of input bits required to obtain a sample.

Huang et al. [HTM17] compile PPs with continuous distributions (but not loops) to MCMC samplers for efficient approximate inference. MCMC algorithms generally provide better inference performance than Zar (which employs an "ordinary Monte Carlo" (OMC) strategy), but suffer from reliability issues (see Chapter 1). Zar is, to our knowledge, the only *formally verified* compiler for PPs with loops and conditioning.

Verified Probabilistic Systems Wang et al. [WHR21] implement a type system based on the notion of *guide types* to guarantee compatibility between model and guide functions in a PPL that compiles to Pyro [BCJ+19]. Pyro is more versatile than Zar, as it supports continuous distributions and programmable inference, but provides no formal guarantees on correctness of compilation or inference. Selsam et al. [SLD18] implement Certigrad, a

PP system for stochastic optimization with formal correctness guarantees in the Lean [MKA+15] theorem prover, but which does not support conditioning or inference.

The Conditional Probabilistic Guarded Command Language (cpGCL) The cpGCL and its corresponding cwp semantics were introduced by Olmedo et al. [OGJ+18] and further developed by Kaminski [Kam19] (including discussion of nondeterminism and expected running time) and Szymcak and Katoen [SK20] (adding support for continuous distributions). These works focus on using cwp as a program logic for verifying individual programs and metatheoretical properties of cpGCL, in contrast to Zar which focuses on verification of compilation to executable samplers.

Interaction Trees Interaction trees have been used to verify compilation of an imperative programming language [XZH+20], networked servers [KLL+19, LRG20], an HTTP key-value server [ZHK+21], and transactional objects [LXK+22]. The Zar system presents a novel application of interaction trees to verified executable semantics of probabilistic programs, and employs a novel domain-theoretic framework for reasoning about them (Section 4.5).

Empirical Evaluation of Probabilistic Programming Systems Dutta et al. [DLHM18] implement a PPL testing framework called ProbFuzz. ProbFuzz generates random test programs for various PPLs and compares their inference results to detect irregularities. We expect that Zar could be incorporated into ProbFuzz as a reference implementation against which the accuracy other discrete PPLs is evaluated.

9.2 AlgCo

This section discusses works related to the proof framework of Algebraic Coinductives described in Chapters 6 and 7.

Parameterized Coinduction (paco) The paco library [HNDV13] provides the most well known generalization of primitive coinduction in Coq. Paco replaces the syntactic guardedness condition for coinductive proofs with a semantic one, leading to a greater degree of compositionality in comparison to Coq's built in cofix mechanism, and alleviation of some common pitfalls, e.g., misapplication of coinduction hypotheses that are not checked until QED-time. Coinductive properties defined using paco are still, however, defined as *greatest fixed points* of monotone functionals, and thus suffer from the problems described in the introduction when attempting to define functional mappings over coinductive structures. With AlgCo, we take an entirely different approach by carving out a subset of coinductive types (those forming algebraic CPOs) for which coinductive reasoning can be replaced by inductive reasoning. The cawu framework of Pous [Pou16] generalizes the theory of parameterized coinduction and simplifies its treatment of "up-to" [Mil89, San98] enhancements of bisimulation, but fundamentally stands in the same relation as paco to the present work.

Copatterns Abel et al. [APTS13] present a type-theoretic foundation for programming with infinite data such as streams via observations (i.e., *copattern* matching). As the categorical dual to inductive pattern matching, which provides a general *elimination* scheme for inductive data, copattern matching provides a general *introduction* scheme for coinductive data. AlgCo fills a gap in the middle by providing a *continuous elimination scheme for algebraic coinductives* (Lemma 5). For example, the function $\mathsf{sum} : \mathsf{Stream}\, \mathbb{R}^{\infty}_{\geq 0} \to \mathbb{R}^{\infty}_{\geq 0}$ described in Section 6.1 can't be defined via copatterns because the codomain is not coinductive, but it *can* be defined as a continuous extension because the domain is algebraic.

Regular Coinductives Jeannin et al. [JKS17] extend OCaml with support for "regular" (finitely representable) coinductives, and equip them with a general elimination scheme

parameterized by user-specified solvers, allowing the programmer to define mappings from regular coinductive structures into CPOs (e.g., mapping regular infinitary λ terms to sets of their free variables via a fixpoint solver). Many coinductive structures we are interested in, however, such as the cotree samplers generated from probabilistic programs described in Section 7.3), are not always finitely representable. The notion of algebraic CPO generalizes to a broader class of coinductive structures while still providing a general (albeit less computable) principle for elimination.

Friendly Functions and Sized Types Syntactic guardedness conditions on primitive corecursive definitions are relaxed in the Isabelle/HOL proof assistant via so-called "friendly functions" [BBL+17], which preserve productivity of their arguments. Corecursive calls are thus allowed to appear under applications of friendly functions. Sized types [HPS96] have been implemented in the Agda proof assistant [Tea22] to fulfill a similar role in broadening the scope of allowable corecursive definitions, and have been used to implement the concatenation and closure operators on coinductive tries [Tra15] as in Section 6.8.1.

Continuous Extensions Lochbihler and Hölzl [LH14] employ concepts from domain theory and topology to define the filter operation on streams as a continuous extension (the "consumer view") of a function on finite lists, and reduce proofs about it to induction over lists. AlgCo does not explicitly require any concepts from topology and generalizes from stream transformers to a wide variety of continuous mappings over coinductive structures including real-valued semantics (Section 7.3) and propositional functions (Section 6.5).

Rusu and Nowak take a similar approach for defining corecursive functions based on finite approximations of CPO elements [RN22] to define filter and mirror operations on streams and rose trees, respectively. We identify the underlying abstraction of algebraic

CPO and provide a principled and coherent story around the composition, verification, and execution of such functions within the AlgCo framework.

10 Conclusion

10.1 Achievements of This Dissertation

This dissertation presents the first formalization of cpGCL and its cwp semantics in a proof assistant, and implements Zar, the first formally verified compiler from a discrete PPL to proved-correct executable samplers. Zar uses a novel intermediate representation, CF trees, to optimize and debias probabilistic choices. CF trees are compiled to executable interaction trees encoding the sampling semantics of source programs in the random bit model. The full compilation pipeline is formally proved to satisfy an equidistribution theorem showing that the empirical distribution of generated samples converges to the true posterior distribution of the source cpGCL program. Zar's backend supports extraction to OCaml and Python and has been used to generate and run samplers for a collection of probabilistic programs including the discrete Gaussian distribution.

We also present AlgCo (Algebraic Coinductives), a formal type-theoretic framework for reasoning about continuous functions over the class of coinductive structures that form algebraic CPOs. We introduce the basic concepts of the framework and provide a Coq implementation in the AlgCo library. We demonstrate its usefulness by implementing and verifying a number of illustrative examples including a stream variant of the sieve of Eratosthenes, a regular expression library based on Brzozowski derivatives, and finally to weakest pre-expectation reasoning over discrete distribution samplers in the random bit model, enabling the verification of coinductive sampling processes compiled by Zar. While the theory of algebraic CPOs has been well known to domain theorists [Gun92], the practical implications of its connection to coinductive types have gone under-appreciated. We are not aware of any prior work to explicitly draw this connection and elaborate on its applications in a theorem proving environment.

10.2 Directions for Future Work

Here we discuss possible directions for extension of the work presented in this dissertation.

Continuous Distributions The cpGCL (and thus the Zar system) supports only discrete posterior probability distributions over program states. It may be feasible to work with a variant of cpGCL in which the binary probabilistic choice command is replaced with a command to generate a single real number uniformly at random from the unit interval $[0, 1]$ *lazily* via its binary expansion representation, one bit at a time. So long as we constrain ourselves to continuous operations over lazily generated reals (perhaps using the machinery of AlgCo), output samples can be obtained up to any desired precision given enough random bits from the sampling environment.

More Efficient Sampling The samplers generated by Zar employ a rejection sampling scheme to produce samples that are independent and identically distributed (i.i.d.). The downside of this approach is that performance may suffer when the probability of rejection is high. Approximating the posterior distribution of a cpGCL program via Monte Carlo techniques with samplers produced by Zar may therefore be infeasible for programs that condition on low-likelihood observations (since observation failure is implemented by rejection in the sampling semantics). The most common technique for improving scalability of approximate sampling-based inference is Markov Chain Monte Carlo [Gey11] sampling, particularly the Metropolis-Hastings and Gibbs sampling algorithms. MCMC samplers produce sequences of samples that are not necessarily i.i.d. but which are distributed in the long run according to the desired posterior, and are able to generate a large number of samples more quickly than alternative methods. Extending the Zar system to generate MCMC samplers would likely provide a dramatic performance improvement for approximate posterior inference of cpGCL programs. We also note that

equidistribution theorem statement (Theorem 10) is agnostic to the implementation details of the sampler and so in principle could apply equally well to an MCMC backend.

Other Methods of Approximate Inference The wp semantics on cotrees (Definition 108) is defined as the supremum of a sequence of approximations tending toward the true probability from below. This leads to a straightforward algorithm for approximating the posterior probability of a given predicate over the sample space by simply computing successive approximations up to the desired precision. In some cases this approach may be more efficient than sampling-based inference, especially when one only cares to determine a lower bound on the probability of an event.

References

[Adá21] Jirí Adámek. Algebraic cocompleteness and finitary functors. *Log. Methods Comput. Sci.*, 17(2), 2021. URL: https://lmcs.episciences.org/7524

[AFG+14] Diego F. Aranha, Pierre-Alain Fouque, Benoît Gérard, Jean-Gabriel Kammerer, Mehdi Tibouchi, and Jean-Christophe Zapalowicz. GLV/GLS decomposition, power analysis, and attacks on ECDSA signatures with single-bit nonce bias. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014. Proceedings, Part I*, volume 8873 of *Lecture Notes in Computer Science*, pages 262–281. Springer, 2014. URL: https://doi.org/10.1007/978-3-662-45611-8_14, doi:10.1007/978-3-662-45611-8_14

[Ahm04] Amal Jamil Ahmed. *Semantics of types for mutable state*. PhD thesis, Princeton University, 2004. TR-713-04. URL: https://www.cs.princeton.edu/techreports/2004/713.pdf

[AJ94] Samson Abramsky and Achim Jung. Domain theory. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3, pages 1–168. Clarendon Press, 1994.

[AK87] Randy Allen and Ken Kennedy. Automatic translation of fortran programs to vector form. *ACM Trans. Program. Lang. Syst.*, 9(4):491–542, 1987. URL: https://doi.org/10.1145/29873.29875, doi:10.1145/29873.29875

[Ake78] Sheldon B. Akers. Binary decision diagrams. *IEEE Trans. Computers*, 27(6):509–516, 1978. URL: https://doi.org/10.1109/TC.1978.1675141, doi:10.1109/TC.1978.1675141

[AL21] Andrew W. Appel and Xavier Leroy. Efficient extensional binary tries. *CoRR*, abs/2110.05063, 2021. URL: https://arxiv.org/abs/2110.05063, arXiv:2110.05063

[ANT+20] Diego F. Aranha, Felipe Rodrigues Novaes, Akira Takahashi, Mehdi Tibouchi, and Yuval Yarom. Ladderleak: Breaking ECDSA with less than one bit of nonce leakage. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, pages 225–242. ACM, 2020. URL: https://doi.org/10.1145/3372297.3417268, doi:10.1145/3372297.3417268

[APTS13] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. Copatterns: programming infinite structures by observations. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 27–38. ACM, 2013. URL: https://doi.org/10.1145/2429069.2429075, doi:10.1145/2429069.2429075

[Arm85] J. Scott Armstrong. *Long-Range Forecasting: From Crystal Ball to Computer*. John Wiley & Sons, New York, 1985.

[BBL+17] Jasmin Christian Blanchette, Aymeric Bouzy, Andreas Lochbihler, Andrei Popescu, and Dmitriy Traytel. Friends with benefits - implementing corecursion in foundational proof assistants. In Hongseok Yang, editor, *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10201 of *Lecture Notes in Computer Science*, pages 111–140. Springer, 2017. URL: https://doi.org/10.1007/978-3-662-54434-1_5, doi:10.1007/978-3-662-54434-1_5

[BC85] Joseph L. Bates and Robert L. Constable. Proofs as programs. *ACM Trans. Program. Lang. Syst.*, 7(1):113–136, 1985. URL: https://doi.org/10.1145/2363.2528, doi:10.1145/2363.2528

[BCJ+19] Eli Bingham, Jonathan P Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D Goodman. Pyro: Deep universal probabilistic programming. *The Journal of Machine Learning Research*, 20(1):973–978, 2019.

[Ber05] Yves Bertot. Filters on coinductive streams, an application to eratosthenes' sieve. In *International Conference on Typed Lambda Calculi and Applications*, pages 102–115. Springer, 2005.

[Ber06] Yves Bertot. Coinduction in coq. *arXiv preprint cs/0603119*, 2006.

[BG22] Verónica Becher and Serge Grigorieff. Randomness and uniform distribution modulo one. *Inf. Comput.*, 285(Part):104857, 2022. URL: https://doi.org/10.1016/j.ic.2021.104857, doi:10.1016/j.ic.2021.104857

[BGJM11] Steve Brooks, Andrew Gelman, Galin Jones, and Xiao-Li Meng. *Handbook of markov chain monte carlo*. CRC press, 2011.

[BH19] Joachim Breitner and Nadia Heninger. Biased nonce sense: Lattice attacks against weak ECDSA signatures in cryptocurrencies. In Ian Goldberg and Tyler Moore, editors, *Financial Cryptography and Data Security - 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18-22, 2019, Revised Selected Papers*, volume 11598 of *Lecture Notes in Computer Science*, pages 3–20. Springer, 2019. URL: https://doi.org/10.1007/978-3-030-32101-7_1, doi:10.1007/978-3-030-32101-7_1

[Bil04] Jeffrey A Bilmes. Graphical models and automatic speech recognition. In *Mathematical foundations of speech and language processing*, pages 191–245. Springer, 2004.

[BK08] Yves Bertot and Vladimir Komendantsky. Fixed point semantics and partial recursion in coq. In Sergio Antoy and Elvira Albert, editors, *Proceedings of the 10th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 15-17, 2008, Valencia, Spain*, pages 89–96. ACM, 2008. URL: https://doi.org/10.1145/1389449.1389461, doi:10.1145/1389449.1389461

[Blu98] Lenore Blum. *Complexity and real computation*. Springer, 1998. URL: https://www.worldcat.org/oclc/37004484

[Bou18] Simon Pierre Boulier. *Extending type theory with syntactic models. (Etendre la théorie des types à l'aide de modèles syntaxiques)*. PhD thesis, Ecole nationale supérieure Mines-Télécom Atlantique Bretagne Pays de la Loire, France, 2018. URL: https://tel.archives-ouvertes.fr/tel-02007839

[BR07] Vladimir Igorevich Bogachev and Maria Aparecida Soares Ruas. *Measure theory*, volume 1. Springer, 2007.

[Bro13] Luitzen Egbertus Jan Brouwer. Intuitionism and formalism. *Bulletin of the American Mathematical Society*, 20(2):81–96, 1913.

[BSB20] Alexander Bagnall, Gordon Stewart, and Anindya Banerjee. Coinductive trees for exact inference of probabilistic programs. In *Languages for Inference (LAFI)*, 2020.

[BT11] George E. .P. Box and George C. Tiao. *Bayesian inference in statistical analysis*. John Wiley & Sons, 2011.

[Cap22] Jane Street Capital. pythonlib, 2022. URL: https://github.com/janestreet/pythonlib

[CD08] Mark Chavira and Adnan Darwiche. On probabilistic inference by weighted model counting. *Artificial Intelligence*, 172(6-7):772–799, 2008.

[CGH+17] Bob Carpenter, Andrew Gelman, Matthew D Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. Stan: A probabilistic programming language. *Journal of statistical software*, 76(1), 2017.

[CH88] Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3):95–120, 1988. URL: https://doi.org/10.1016/0890-5401(88)90005-3, doi:10.1016/0890-5401(88)90005-3

[Cha10] Arthur Charguéraud. The optimal fixed point combinator. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6172 of *Lecture Notes in Computer Science*, pages 195–210. Springer, 2010. URL: https://doi.org/10.1007/978-3-642-14052-5_15, doi:10.1007/978-3-642-14052-5_15

[Cha17] Arthur Charguéraud. Coqandaxioms, 2017. URL: https://github.com/coq/coq/wiki/CoqAndAxioms

[Cha23] Arthur Charguéraud. Tlc: a non-constructive library for coq, 2023. URL: https://www.chargueraud.org/softs/tlc/

[Chi65] Charles S Chihara. On the possibility of completing an infinite process. *The Philosophical Review*, 74(1):74–87, 1965.

[Chl22] Adam Chlipala. *Certified programming with dependent types: a pragmatic introduction to the Coq proof assistant*. MIT Press, 2022.

[Chu28] Alonzo Church. On the law of excluded middle. *Bulletin of the American Mathematical Society*, 34(1):75–78, 1928.

[CKS20] Clément L Canonne, Gautam Kamath, and Thomas Steinke. The discrete gaussian for differential privacy. *Advances in Neural Information Processing Systems*, 33:15676–15688, 2020.

[CMS22] David Chiang, Colin McDonald, and Chung-chieh Shan. Exact recursive probabilistic programming. *arXiv preprint arXiv:2210.01206*, 2022.

[com22] The SciPy community. scipy.stats, 2022. URL: https://docs.scipy.org/doc/scipy/reference/stats.html

[Coo90] Gregory F. Cooper. The computational complexity of probabilistic inference using bayesian belief networks. *Artif. Intell.*, 42(2-3):393–405, 1990. URL: https://doi.org/10.1016/0004-3702(90)90060-D, doi:10.1016/0004-3702(90)90060-D

[coq23a] Library coq.logic.classicalepsilon, 2023. URL: https://coq.inria.fr/library/Coq.Logic.ClassicalEpsilon.html

[coq23b] Library coq.reals.reals, 2023. URL: https://coq.inria.fr/library/Coq.Reals.Reals.html

[CRN+13] Guillaume Claret, Sriram K Rajamani, Aditya V Nori, Andrew D Gordon, and Johannes Borgström. Bayesian inference using data flow analysis. In *Proceedings of the 2013 9th joint meeting on foundations of software engineering*, pages 92–102, 2013.

[CTSLM19] Marco F Cusumano-Towner, Feras A Saad, Alexander K Lew, and Vikash K Mansinghka. Gen: a general-purpose probabilistic programming system with programmable inference. In *Proceedings of the 40th acm sigplan conference on programming language design and implementation*, pages 221–236, 2019.

[CTY06] Nick Chater, Joshua B Tenenbaum, and Alan Yuille. Probabilistic models of cognition: Conceptual foundations. *Trends in cognitive sciences*, 10(7):287–291, 2006.

[Cur34] Haskell B Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences*, 20(11):584–590, 1934.

[Dev86] Luc Devroye. *Non-Uniform Random Variate Generation*. Springer, 1986. URL: https://doi.org/10.1007/978-1-4613-8643-8, doi:10.1007/978-1-4613-8643-8

[DG04] Rodney G Downey and Evan J Griffiths. Schnorr randomness. *The Journal of Symbolic Logic*, 69(2):533–554, 2004.

[Dij75] Edsger W Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[DLHM18] Saikat Dutta, Owolabi Legunsen, Zixin Huang, and Sasa Misailovic. Testing probabilistic programming systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 574–586, 2018.

[DM02] Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002.

[DZHM19] Saikat Dutta, Wenxian Zhang, Zixin Huang, and Sasa Misailovic. Storm: program reduction for testing and debugging probabilistic programming

systems. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 729–739, 2019.

[Elk21] Derek Elkins. Tying the knot, 2021. URL: https://wiki.haskell.org/Tying_the_Knot

[FNSB15] Alireza Farasat, Alexander Nikolaev, Sargur N Srihari, and Rachael Hageman Blair. Probabilistic graphical models in modern social network analysis. *Social Network Analysis and Mining*, 5(1):62, 2015.

[Fri04] Nir Friedman. Inferring cellular networks using probabilistic graphical models. *Science*, 303(5659):799–805, 2004.

[GCST19] Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. Definitional proof-irrelevance without k. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–28, 2019.

[Gey11] Charles Geyer. Introduction to markov chain monte carlo. *Handbook of Markov Chain Monte Carlo*, 20116022:45, 2011.

[GMR+12] Noah Goodman, Vikash Mansinghka, Daniel M Roy, Keith Bonawitz, and Joshua B Tenenbaum. Church: a language for generative models. *arXiv preprint arXiv:1206.3255*, 2012.

[Gro23] Jason Gross. Coinductive extensionality, 2023. URL: https://stackoverflow.com/a/69905520

[GRS09] Arpita Ghosh, Tim Roughgarden, and Mukund Sundararajan. Universally utility-maximizing privacy mechanisms. In Michael Mitzenmacher, editor, *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009*, pages 351–360. ACM, 2009. URL: https://doi.org/10.1145/1536414.1536464, doi:10.1145/1536414.1536464

[GS14] Noah D Goodman and Andreas Stuhlmüller. The design and implementation of probabilistic programming languages, 2014.

[Gun92] Carl A Gunter. *Semantics of programming languages: structures and techniques*. MIT press, 1992.

[H+56] Thomas Little Heath et al. *The thirteen books of Euclid's Elements*. Courier Corporation, 1956.

[Hag89] Tatsuya Hagino. Codatatypes in ML. *J. Symb. Comput.*, 8(6):629–650, 1989. URL: https://doi.org/10.1016/S0747-7171(89)80065-3, doi:10.1016/S0747-7171(89)80065-3

[Hal13] Paul R Halmos. *Measure theory*, volume 18 of *Graduate Texts in Mathematics*. Springer, 2013. Originally published by Litton Educational Publishing, Inc., 1950. URL: https://doi.org/10.1007/978-1-4684-9440-2, doi:10.1007/978-1-4684-9440-2

[HdBM20] Steven Holtzen, Guy Van den Broeck, and Todd D. Millstein. Scaling exact inference for discrete probabilistic programs. *Proc. ACM Program. Lang.*, 4(OOPSLA):140:1–140:31, 2020. URL: https://doi.org/10.1145/3428208, doi:10.1145/3428208

[HMB19] Steven Holtzen, Todd Millstein, and Guy Van den Broeck. Symbolic exact inference for discrete probabilistic programs. *arXiv preprint arXiv:1904.02079*, 2019.

[HNDV13] Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. The power of parameterization in coinductive proof. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 193–206. ACM, 2013. URL: https://doi.org/10.1145/2429069.2429093, doi:10.1145/2429069.2429093

[How80] William A Howard. The formulae-as-types notion of construction. *To HB Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, 44:479–490, 1980.

[HPS96] John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In Hans-Juergen Boehm and Guy L. Steele Jr., editors, *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, pages 410–423. ACM Press, 1996. URL: https://doi.org/10.1145/237721.240882, doi:10.1145/237721.240882

[HTM17] Daniel Huang, Jean-Baptiste Tristan, and Greg Morrisett. Compiling markov chain monte carlo algorithms for probabilistic modeling. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 111–125. ACM, 2017. URL: https://doi.org/10.1145/3062341.3062375, doi:10.1145/3062341.3062375

[INR22] INRIA. Coq reference manual, 2022. URL: https://coq.inria.fr/refman/#

[INR23] INRIA. Typing rules, 2023. URL: https://coq.inria.fr/distrib/current/refman/language/cic.html

[Isa03] Michael Isard. Pampas: Real-valued graphical models for computer vision. In *2003 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2003. Proceedings.*, volume 1, pages I–I. IEEE, 2003.

[JD93] Mark P Jones and Luc Duponcheel. Composing monads. Technical report, Technical Report YALEU/DCS/RR-1004, Department of Computer Science. Yale . . . , 1993.

[JKS17] Jean-Baptiste Jeannin, Dexter Kozen, and Alexandra Silva. Cocaml: Functional programming with regular coinductive types. *Fundam. Informaticae*, 150(3-4):347–377, 2017. URL: https://doi.org/10.3233/FI-2017-1473, doi:10.3233/FI-2017-1473

[Jun90] Achim Jung. Cartesian closed categories of algebraic cpos. *Theor. Comput. Sci.*, 70(2):233–250, 1990. URL: https://doi.org/10.1016/0304-3975(90)90124-Z, doi:10.1016/0304-3975(90)90124-Z

[Kam19] Benjamin Lucien Kaminski. *Advanced weakest precondition calculi for probabilistic programs*. PhD thesis, RWTH Aachen University, Germany, 2019. URL: http://publications.rwth-aachen.de/record/755408

[Kec12] Alexander Kechris. *Classical descriptive set theory*, volume 156. Springer Science & Business Media, 2012.

[KF09] Daphne Koller and Nir Friedman. *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

[KL51] Solomon Kullback and Richard A Leibler. On information and sufficiency. *The annals of mathematical statistics*, 22(1):79–86, 1951.

[Kle52] Stephen Cole Kleene. Introduction to metamathematics. 1952.

[KLL+19] Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, William Mansky, Benjamin C Pierce, and Steve Zdancewic. From c to interaction trees: specifying, verifying, and testing a networked server. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 234–248, 2019.

[KN12] Lauwerens Kuipers and Harald Niederreiter. *Uniform distribution of sequences*. Courier Corporation, 2012.

[KN15] Mahmoud Khademi and Nedialko S Nedialkov. Probabilistic graphical models and deep belief networks for prognosis of breast cancer. In *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*, pages 727–732. IEEE, 2015.

[Koz85] Dexter Kozen. A probabilistic PDL. *J. Comput. Syst. Sci.*, 30(2):162–178, 1985. URL: https://doi.org/10.1016/0022-0000(85)90012-1, doi:10.1016/0022-0000(85)90012-1

[KS17] Dexter Kozen and Alexandra Silva. Practical coinduction. *Mathematical Structures in Computer Science*, 27(7):1132–1152, 2017.

[KY76] Donald E. Knuth and Andrew C. Yao. The complexity of nonuniform random number generation. 1976.

[LBBH98] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[LDF+21] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The OCaml system release 4.13: Documentation and user's manual. Intern report, Inria, September 2021. URL: https://hal.inria.fr/hal-00930213

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[Ler15] Xavier Leroy. Using coq's evaluation mechanisms in anger, 2015. URL: http://gallium.inria.fr/blog/coq-eval/

[Let08] Pierre Letouzey. Extraction in coq: An overview. In *Conference on Computability in Europe*, pages 359–369. Springer, 2008.

[LH14] Andreas Lochbihler and Johannes Hölzl. Recursive functions on lazy lists via domains and topologies. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8558 of *Lecture Notes in Computer Science*, pages 341–357. Springer, 2014. URL: https://doi.org/10.1007/978-3-319-08970-6_22, doi:10.1007/978-3-319-08970-6_22

[LRG20] Thomas Letan and Yann Régis-Gianas. Freespec: specifying, verifying, and executing impure computations in coq. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 32–46, 2020.

[LXK+22] Mohsen Lesani, Li-yao Xia, Anders Kaseorg, Christian J. Bell, Adam Chlipala, Benjamin C. Pierce, and Steve Zdancewic. C4: Verified transactional objects. *Proc. ACM Program. Lang.*, 6(OOPSLA):1–31, 2022. URL: https://doi.org/10.1145/3527324, doi:10.1145/3527324

[Mar84] Per Martin-Löf. *Intuitionistic type theory*, volume 1 of *Studies in proof theory*. Bibliopolis, 1984.

[McE20] Richard McElreath. *Statistical rethinking: A Bayesian course with examples in R and Stan*. Chapman and Hall/CRC, 2020.

[MFP91] Erik Meijer, Maarten M. Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In John Hughes, editor, *Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer, 1991. URL: https://doi.org/10.1007/3540543961_7, doi:10.1007/3540543961_7

[Mil89] Robin Milner. *Communication and concurrency*, volume 84. Prentice hall Englewood Cliffs, 1989.

[MKA+15] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In *International Conference on Automated Deduction*, pages 378–388. Springer, 2015.

[ML66] Per Martin-Löf. The definition of random sequences. *Information and control*, 9(6):602–619, 1966.

[ML13] Saunders Mac Lane. *Categories for the working mathematician*, volume 5. Springer Science & Business Media, 2013.

[MM05] Annabelle McIver and Carroll Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems*. Monographs in Computer Science. Springer, 2005. URL: https://doi.org/10.1007/b138392, doi:10.1007/b138392

[MMS96] Carroll Morgan, Annabelle McIver, and Karen Seidel. Probabilistic predicate transformers. *ACM Trans. Program. Lang. Syst.*, 18(3):325–353, 1996. URL: https://doi.org/10.1145/229542.229547, doi:10.1145/229542.229547

[Ngu04] Phong Q. Nguyen. Can we trust cryptographic software? cryptographic flaws in GNU privacy guard v1.2.3. In Christian Cachin and Jan Camenisch, editors, *Advances in Cryptology - EUROCRYPT 2004, International Conference on the Theory and Applications of Cryptographic Techniques, Interlaken, Switzerland, May 2-6, 2004, Proceedings*, volume 3027 of *Lecture Notes in Computer Science*, pages 555–570. Springer, 2004. URL: https://doi.org/10.1007/978-3-540-24676-3_33, doi:10.1007/978-3-540-24676-3_33

[OGJ+18] Federico Olmedo, Friedrich Gretz, Nils Jansen, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Annabelle McIver. Conditioning in probabilistic programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 40(1):1–50, 2018.

[OKKM16] Federico Olmedo, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. Reasoning about recursive probabilistic programs. In *2016 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–10. IEEE, 2016.

[O'N09] Melissa E. O'Neill. The genuine sieve of eratosthenes. *J. Funct. Program.*, 19(1):95–106, 2009. URL: https://doi.org/10.1017/S0956796808007004, doi:10.1017/S0956796808007004

[PA19] Daniel Patterson and Amal Ahmed. The next 700 compiler correctness theorems (functional pearl). *Proc. ACM Program. Lang.*, 3(ICFP):85:1–85:29, 2019. URL: https://doi.org/10.1145/3341689, doi:10.1145/3341689

[PCG+10] Benjamin C Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. Software foundations. *Webpage: http://www. cis. upenn. edu/bcpierce/sf/current/index. html*, 2010.

[PD07] Hoifung Poon and Pedro Domingos. Joint inference in information extraction. In *AAAI*, volume 7, pages 913–918, 2007.

[Pfe01] Avi Pfeffer. Ibal: A probabilistic rational programming language. In *IJCAI*, pages 733–740. Citeseer, 2001.

[PGJ16] Judea Pearl, Madelyn Glymour, and Nicholas P Jewell. *Causal inference in statistics: A primer*. John Wiley & Sons, 2016.

[Pie91] Benjamin C. Pierce. *Basic category theory for computer scientists*. Foundations of computing. MIT Press, 1991.

[Pie16] Benjamin C. Pierce. The science of deep specification (keynote). In Eelco Visser, editor, *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity, SPLASH 2016, Amsterdam, Netherlands, October 30 - November 4, 2016*, page 1. ACM, 2016. URL: https://doi.org/10.1145/2984043.2998388, doi:10.1145/2984043.2998388

[Pou16] Damien Pous. Coinduction all the way up. In Martin Grohe, Eric Koskinen, and Natarajan Shankar, editors, *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New*

York, NY, USA, July 5-8, 2016, pages 307–316. ACM, 2016. URL: https://doi.org/10.1145/2933575.2934564, doi:10.1145/2933575.2934564

[Put00] Tumkur K Puttaswamy. The mathematical accomplishments of ancient indian mathematicians. In *Mathematics Across Cultures*, pages 409–422. Springer, 2000.

[R+76] Walter Rudin et al. *Principles of mathematical analysis*, volume 3. McGraw-hill New York, 1976.

[RK16] Reuven Y Rubinstein and Dirk P Kroese. *Simulation and the Monte Carlo method*. John Wiley & Sons, 2016.

[RM19] Sebastian Raschka and Vahid Mirjalili. *Python machine learning: Machine learning and deep learning with Python, scikit-learn, and TensorFlow 2*. Packt Publishing Ltd, 2019.

[RN22] Vlad Rusu and David Nowak. Defining corecursive functions in Coq using approximations. In Karim Ali and Jan Vitek, editors, *36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany*, volume 222 of *LIPIcs*, pages 12:1–12:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. URL: https://doi.org/10.4230/LIPIcs.ECOOP.2022.12, doi:10.4230/LIPIcs.ECOOP.2022.12

[RU08] Tom Richardson and Ruediger Urbanke. *Modern coding theory*. Cambridge university press, 2008.

[San98] Davide Sangiorgi. On the bisimulation proof method. *Mathematical Structures in Computer Science*, 8(5):447–479, 1998.

[Sco70] Dana Scott. *Outline of a mathematical theory of computation*. Oxford University Computing Laboratory, Programming Research Group Oxford, 1970.

[SCS+19] Feras A. Saad, Marco F. Cusumano-Towner, Ulrich Schaechtle, Martin C. Rinard, and Vikash K. Mansinghka. Bayesian synthesis of probabilistic programs for automatic data modeling. *Proc. ACM Program. Lang.*, 3(POPL):37:1–37:32, 2019. URL: https://doi.org/10.1145/3290350, doi:10.1145/3290350

[Sec20] Kudelski Security. The definitive guide to ”modulo bias and how to avoid it”!, 2020. URL: https://research.kudelskisecurity.com/2020/07/28/the-definitive-guide-to-modulo-bias-and-how-to-avoid-it/

[SFRM20a] Feras Saad, Cameron E. Freer, Martin C. Rinard, and Vikash Mansinghka. The fast loaded dice roller: A near-optimal exact sampler for discrete probability distributions. In Silvia Chiappa and Roberto Calandra, editors, *The 23rd International Conference on Artificial Intelligence and Statistics, AISTATS 2020, 26-28 August 2020, Online [Palermo, Sicily, Italy]*, volume 108 of *Proceedings of Machine Learning Research*, pages 1036–1046. PMLR, 2020. URL: http://proceedings.mlr.press/v108/saad20a.html

[SFRM20b] Feras A. Saad, Cameron E. Freer, Martin C. Rinard, and Vikash K. Mansinghka. Optimal approximate sampling from discrete probability distributions. *Proc. ACM Program. Lang.*, 4(POPL):36:1–36:31, 2020. URL: https://doi.org/10.1145/3371104, doi:10.1145/3371104

[Sha48] Claude Elwood Shannon. A mathematical theory of communication. *The Bell system technical journal*, 27(3):379–423, 1948.

[SK20] Marcin Szymczak and Joost-Pieter Katoen. Weakest preexpectation semantics for bayesian inference. *CoRR*, abs/2005.09013, 2020. URL: https://arxiv.org/abs/2005.09013, arXiv:2005.09013

[SLD18] Daniel Selsam, Percy Liang, and David L Dill. Formal methods for probabilistic programming. In *Workshop on Probabilistic Programming Languages, Semantics, and Systems*. 2018.

[SM16] Feras Saad and Vikash Mansinghka. Probabilistic data analysis with probabilistic programming. *CoRR*, abs/1608.05347, 2016. URL: http://arxiv.org/abs/1608.05347, arXiv:1608.05347

[Soz09] Matthieu Sozeau. A new look at generalized rewriting in type theory. *J. Formaliz. Reason.*, 2(1):41–62, 2009. URL: https://doi.org/10.6092/issn.1972-5787/1574, doi:10.6092/issn.1972-5787/1574

[SRM21] Feras A. Saad, Martin C. Rinard, and Vikash K. Mansinghka. SPPL: probabilistic programming with fast exact symbolic inference. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 804–819. ACM, 2021. URL: https://doi.org/10.1145/3453483.3454078, doi:10.1145/3453483.3454078

[Tea22] The Agda Team. Agda user manual, 2022. URL: https://agda.readthedocs.io/en/v2.6.2.2.20221128/

[Tho13] Steve Thomas. Decryptocat, 2013. URL: https://tobtu.com/decryptocat.php

[Tra15] Dmytro Traytel. *Formalizing Symbolic Decision Procedures for Regular Languages*. PhD thesis, Technical University Munich, 2015. URL: https://nbn-resolving.org/urn:nbn:de:bvb:91-diss-20151016-1273011-1-9

[TvdMYW16] David Tolpin, Jan Willem van de Meent, Hongseok Yang, and Frank Wood. Design and implementation of probabilistic programming language anglican. *arXiv preprint arXiv:1608.05263*, 2016.

[vdMPYW18] Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. An introduction to probabilistic programming. *CoRR*, abs/1809.10756, 2018. URL: http://arxiv.org/abs/1809.10756, arXiv:1809.10756

[VF45] Kurt Von Fritz. The discovery of incommensurability by Hippasus of Metapontum. *Annals of mathematics*, pages 242–264, 1945.

[VN51] John Von Neumann. Various techniques used in connection with random digits. *Appl. Math Ser*, 12(36-38):5, 1951.

[Wad92] Philip Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14, 1992.

[Wey16] Hermann Weyl. Über die gleichverteilung von zahlen mod. eins. *Mathematische Annalen*, 77(3):313–352, 1916.

[WHR21] Di Wang, Jan Hoffmann, and Thomas W. Reps. Sound probabilistic inference via guide types. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 788–803. ACM, 2021. URL: https://doi.org/10.1145/3453483.3454077, doi:10.1145/3453483.3454077

[WW11] Shen SJ Wang and Matt P Wand. Using infer. net for statistical analyses. *The American Statistician*, 65(2):115–126, 2011.

[Xia23] Li-yao Xia. Interaction trees, 2023. URL: https://github.com/DeepSpec/InteractionTrees

[XZH+20] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. Interaction trees: representing recursive and impure programs in coq. *Proc. ACM Program. Lang.*, 4(POPL):51:1–51:32, 2020. URL: https://doi.org/10.1145/3371119, doi:10.1145/3371119

[YFW03] Jonathan S Yedidia, William T Freeman, and Yair Weiss. Understanding belief propagation and its generalizations. *Exploring artificial intelligence in the new millennium*, 8:236–239, 2003.

[ZHK+21] Hengchu Zhang, Wolf Honoré, Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. Verifying an HTTP key-value server with interaction trees and VST. In Liron Cohen and Cezary Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference)*, volume 193 of *LIPIcs*, pages 32:1–32:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. URL: https://doi.org/10.4230/LIPIcs.ITP.2021.32, doi:10.4230/LIPIcs.ITP.2021.32

[ZSS20] Raymond K Zhao, Ron Steinfeld, and Amin Sakzad. Cosac: Compact and scalable arbitrary-centered discrete gaussian sampling over integers. In *International Conference on Post-Quantum Cryptography*, pages 284–303. Springer, 2020.

Appendix: Extended Reals

The space $\mathbb{R}$ of real numbers provides an account for the so-called *incommensurable magnitudes* [VF45] such as $\sqrt{2}$ (discovered by followers of Pythagoras, and perhaps even earlier in ancient India [Put00]) via its completeness property: any bounded set of reals has a least upper bound. However, the space $\mathbb{R}$ equipped with its usual ordering is technically *not* a CPO and is thus not quite sufficient for our purposes. The problem is that only *bounded* sets of reals (e.g. the set $\{x \mid x^2 < 2\}$) are guaranteed to have least upper bounds. To ensure the existence of suprema for *all* subsets, the reals must be extended with a "point at infinity" $+\infty$ to serve as the supremum of unbounded sets.

In addition, since this work is only ever concerned with *nonnegative* real numbers we include a nonnegativity constraint, arriving at the space $\mathbb{R}^\infty_{\geq 0}$ of *nonnegative extended reals* defined as follows:

Definition 127 ($\mathbb{R}^\infty_{\geq 0}$)**.** *Define the type $\mathbb{R}^\infty_{\geq 0}$ of nonnegative extended real numbers by the following formation rules:*

$$\frac{r : \mathbb{R} \qquad 0 \leq r}{\mathbf{er}\ r : \mathbb{R}^\infty_{\geq 0}}\ \textsc{eR-er} \qquad\qquad \frac{}{\mathbf{infty} : \mathbb{R}^\infty_{\geq 0}}\ \textsc{eR-infty}$$

We let '**er** a' be denoted by simply 'a' when clear from context (e.g., '**er** 0' denoted by '0', '**er** 1' by '1', etc.). The extended reals are ordered in the usual way and such that $a \leq +\infty$ for all $a : \mathbb{R}^\infty_{\geq 0}$:

Definition 128 ($\leq_{\mathbb{R}^\infty_{\geq 0}}$)**.**

$$\frac{a \leq_{\mathbb{R}} b}{\mathbf{er}\ a \leq_{\mathbb{R}^\infty_{\geq 0}} \mathbf{er}\ b}\ \textsc{eR-le-er} \qquad\qquad \frac{a : \mathbb{R}^\infty_{\geq 0}}{a \leq \mathbf{infty}}\ \textsc{eR-le-infty}$$

Addition, subtraction, multiplication and the multiplicative inverse operator (from which division is derived) are defined as follows:

Definition 129 (eRplus)**.** *For* $a : \mathbb{R}^\infty_{\geq 0}$ *and* $b : \mathbb{R}^\infty_{\geq 0}$*, define* $a + b : \mathbb{R}^\infty_{\geq 0}$ *by cases:*

$$+_{\mathbb{R}^\infty_{\geq 0}} : \mathbb{R}^\infty_{\geq 0} \to \mathbb{R}^\infty_{\geq 0} \to \mathbb{R}^\infty_{\geq 0}$$

$$\begin{aligned}
\textbf{infty} \;+\; _ &\triangleq \textbf{infty} \\
_ \;+\; \textbf{infty} &\triangleq \textbf{infty} \\
\textbf{er}\ a \;+\; \textbf{er}\ b &\triangleq \textbf{er}\ (a + b)
\end{aligned}$$

Definition 130 (eRminus)**.** *For* $a : \mathbb{R}^\infty_{\geq 0}$ *and* $b : \mathbb{R}^\infty_{\geq 0}$*, define* $a - b : \mathbb{R}^\infty_{\geq 0}$ *by cases:*

$$-_{\mathbb{R}^\infty_{\geq 0}} : \mathbb{R}^\infty_{\geq 0} \to \mathbb{R}^\infty_{\geq 0} \to \mathbb{R}^\infty_{\geq 0}$$

$$\begin{aligned}
\textbf{infty} \;-\; \textbf{er}\ _ &\triangleq \textbf{infty} \\
_ \;-\; \textbf{infty} &\triangleq \textbf{er}\ 0 \\
\textbf{er}\ a \;-\; \textbf{er}\ b &\triangleq \textit{if}\ b \leq a\ \textit{then}\ \textbf{er}\ (a - b)\ \textit{else}\ \textbf{er}\ 0
\end{aligned}$$

Definition 131 (eRmult)**.** *For* $a : \mathbb{R}^\infty_{\geq 0}$ *and* $b : \mathbb{R}^\infty_{\geq 0}$*, define* $a \cdot b : \mathbb{R}^\infty_{\geq 0}$ *by cases:*

$$\cdot_{\mathbb{R}^\infty_{\geq 0}} : \mathbb{R}^\infty_{\geq 0} \to \mathbb{R}^\infty_{\geq 0} \to \mathbb{R}^\infty_{\geq 0}$$

$$\begin{aligned}
\textbf{infty} \cdot b &\triangleq \textit{if}\ b = \textbf{er}\ 0\ \textit{then}\ \textbf{er}\ 0\ \textit{else}\ \textbf{infty} \\
a \cdot \textbf{infty} &\triangleq \textit{if}\ a = \textbf{er}\ 0\ \textit{then}\ \textbf{er}\ 0\ \textit{else}\ \textbf{infty} \\
\textbf{er}\ a \cdot \textbf{er}\ b &\triangleq \textbf{er}\ (a \cdot b)
\end{aligned}$$

Definition 132 (eRinv)**.** *For* $a : \mathbb{R}^\infty_{\geq 0}$*, define* $a^{-1} : \mathbb{R}^\infty_{\geq 0}$ *by cases:*

$$\begin{array}{l} \mathsf{eRinv} : \mathbb{R}^\infty_{\geq 0} \to \mathbb{R}^\infty_{\geq 0} \\ \hline \mathbf{infty}^{-1} \triangleq \mathbf{er}\ 0 \\ (\mathbf{er}\ r)^{-1} \triangleq \textit{if } a = 0 \textit{ then } \mathbf{infty} \textit{ else } \mathbf{er}\ a^{-1} \end{array}$$

We follow the convention that multiplication by 0 annihilates all elements, including $+\infty$. That is, we have $0 \cdot a = a \cdot 0 = 0$ for *all* a, even when $a = +\infty$. However, when $a \neq 0$, $+\infty \cdot a = a \cdot +\infty = +\infty$.

Exponentiation and division are defined as follows, where exponentiation is derived via repeated multiplication and division via multiplication by the inverse of the right-hand argument:

Definition 133 (eRpow)**.** *For* $a : \mathbb{R}^\infty_{\geq 0}$ *and* $n : \mathbb{N}$*, define* $a^n : \mathbb{R}^\infty_{\geq 0}$ *by induction on* n*:*

$$\begin{array}{l} \mathsf{eRpow} : \mathbb{R}^\infty_{\geq 0} \to \mathbb{N} \to \mathbb{R}^\infty_{\geq 0} \\ \hline a^0 \triangleq \mathbf{er}\ 1 \\ a^{S\,n} \triangleq a \cdot a^n \end{array}$$

Definition 134 (eRdiv)**.** *For* $a : \mathbb{R}^\infty_{\geq 0}$ *and* $b : \mathbb{R}^\infty_{\geq 0}$*, define* $\dfrac{a}{b} : \mathbb{R}^\infty_{\geq 0} \triangleq a \cdot b^{-1}$.

Proof Automation Many goals involving real numbers can be automatically proved by the lra tactic from Coq's standard library. This is a great luxury as it relieves the burden of handling meticulous details of arithmetic typically skipped over in pencil-and-paper proofs but which are annoyingly labourious to carry out in a formal proof environment.

Unfortunately, by defining our own type $\mathbb{R}^{\infty}_{\geq 0}$ of extended reals we lose access to the automation machinery of lra and similar tactics (and we are not aware at the time of writing of any straightforward way to extend said automation to handle new types). However, we are able to partially compensate for this loss by defining a custom proof tactic ‘eRauto‘ that implements a simple rewrite system for basic identities (e.g., $0 + a = a$) and automatically applies lemmas drawn from a database of commonly useful results (contained in the file ‘eR.v’).

Thesis and Dissertation Services