Deriving a Probability Density Calculator (Functional Pearl)

Wazim Mohammed Ismail Chung-chieh Shan

Indiana University, USA

{wazimoha,ccshan}@indiana.edu

Abstract

Given an expression that denotes a probability distribution, often we want a corresponding *density* function, to use in probabilistic inference. Fortunately, the task of finding a density has been automated. It turns out that we can *derive* a compositional procedure for finding a density, by equational reasoning about integrals, starting with the mathematical specification of what a density is. Moreover, the density found can be run as an estimation algorithm, as well as simplified as an exact formula to improve the estimate.

Categories and Subject Descriptors G.3 [*Probability and Statistics*]: distribution functions; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—specification techniques; D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming

Keywords probability density functions, probability measures, continuations, program calculation, equational reasoning

1. Introduction

A popular way to handle uncertainty in AI, statistics, and science is to compute with probability distributions. Typically, we define a distribution then answer questions about it such as "what is its expected value?" and "what does its histogram look like?". Over a century, practitioners of this approach have identified many patterns in how to define distributions (that is, *modeling*) and how to answer questions about them (called *inference*). These patterns constitute the beginning of a *combinator library* (Hughes 1995).

Unfortunately, models and inference procedures do not compose in tandem: as illustrated in Sections 3.1 and 8.2, often we rejoice that a large distribution we're interested in can be expressed naturally by composing smaller distributions, but then despair that many questions we want to pose about the overall distribution cannot be answered using answers to corresponding questions about the constituent distributions. In other words, the natural compositional structure of models and of inference procedures are not the same. This mismatch is disappointing because it makes it harder for us to automate the labor-intensive process of turning a distribution that models the world into a program that answers relevant questions about it. This difficulty is the bane of declarative programming. It's like trying to build a SAT solver that generates assignments satisfying a compound expression $e_1 \wedge e_2$ by combining assignments satisfying the subexpressions e_1 and e_2.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the Owner/Author.

Copyright is held by the owner/author(s).

ICFP'16, September 18–24, 2016, Nara, Japan
ACM. 978-1-4503-4219-3/16/09...$15.00
http://dx.doi.org/10.1145/2951913.2951922

$$\frac{}{\mathsf{StdRandom} : \mathit{Real}} \qquad \frac{x \in \mathbb{R}}{\mathsf{Lit}\ x : \mathit{Real}} \qquad \frac{e : a \quad \begin{array}{c}\mathsf{Var}\ v : a \\ \vdots \\ e' : b\end{array}}{\mathsf{Let}\ v\ e\ e' : b}$$

$$\frac{e : \mathit{Real}}{\mathsf{Neg}\ e : \mathit{Real}} \qquad \frac{e : \mathit{Real}}{\mathsf{Exp}\ e : \mathit{Real}} \qquad \frac{e : \mathit{Real}}{\mathsf{Log}\ e : \mathit{Real}} \qquad \frac{e : \mathit{Bool}}{\mathsf{Not}\ e : \mathit{Bool}}$$

$$\frac{e_1 : \mathit{Real} \quad e_2 : \mathit{Real}}{\mathsf{Add}\ e_1\ e_2 : \mathit{Real}} \qquad \frac{e_1 : \mathit{Real} \quad e_2 : \mathit{Real}}{\mathsf{Less}\ e_1\ e_2 : \mathit{Bool}} \qquad \frac{e : \mathit{Bool} \quad e_1 : a \quad e_2 : a}{\mathsf{If}\ e\ e_1\ e_2 : a}$$

Figure 1. The type system of our language of distributions

Still, there's hope to answer more inference questions while following the natural compositional structure of models, if only we could figure out how to generalize the questions as if strengthening an induction hypothesis or adding an accumulator argument. This paper tells one such success story. We answer the questions

1. "What is the expected value of this distribution?"
2. "What is a density function of this distribution?"

by generalizing them to compositional interpreters. Our interpreters are compilers in the sense that their output can be simplified using a computer algebra system or executed as a randomized algorithm. We derive these compilers by equational reasoning from a semantic specification. Our derivation appeals to λ-calculus equations alongside integral-calculus equations.

2. A Language of Generative Stories

To be concrete, we define a small language of distributions:

Terms	$e ::= \mathsf{StdRandom} \mid \mathsf{Lit}\ x \mid \mathsf{Var}\ v \mid \mathsf{Let}\ v\ e\ e$
	$\mid \mathsf{Neg}\ e \mid \mathsf{Exp}\ e \mid \mathsf{Log}\ e \mid \mathsf{Not}\ e$
	$\mid \mathsf{Add}\ e\ e \mid \mathsf{Less}\ e\ e \mid \mathsf{If}\ e\ e\ e$
Variables	v
Real numbers	x

Figure 1 shows our type system. To keep things simple, we include only two types in this language, *Real* and *Bool*. As usual, the typing judgment $e : a$ means that the expression e has the type a. In $\mathsf{Let}\ v\ e\ e'$, the bound variable v takes scope over e' and not e.

Each expression says how to generate a random outcome. For example, the atomic expression StdRandom says to choose a random real number uniformly between 0 and 1. That's why its type is *Real*. To take another example, the compound expression

```
Add StdRandom StdRandom
```

says to choose two random real numbers independently, each uniformly between 0 and 1, then sum them. The sum is again a real

number, so this expression's type is also *Real*. These descriptions of how to generate a random outcome are called *generative stories* by applied statisticians, and also called *generators* in QuickCheck (Claessen and Hughes 2000). The intuitive meaning of a generative story is a distribution over its outcomes, such as over reals. Because generative stories are intuitive to tell, and because they make it easy to detect dependencies among random choices (Pearl 1988), it's popular to express probability distributions by composing generative stories—such as using Add. The syntax of our language thus embodies the "natural compositional structure of models" referred to in the introduction above.

We define a Haskell data type *Expr* to encode the expressions of our language. Actually, using the GADT (generalized algebraic data type) extension, let's define two Haskell types *Expr Real* and *Expr Bool* at the same time, to distinguish our types *Real* and *Bool*.

```
data Expr a where
  StdRandom ::                                        Expr Real
  Lit       :: Rational →                             Expr Real
  Var       :: Var a →                                Expr a
  Let       :: Var a → Expr a → Expr b →              Expr b
  Neg, Exp,
      Log   :: Expr Real →                            Expr Real
  Not       :: Expr Bool →                            Expr Bool
  Add       :: Expr Real → Expr Real →                Expr Real
  Less      :: Expr Real → Expr Real →                Expr Bool
  If        :: Expr Bool → Expr a → Expr a →          Expr a
```

(Although we can easily construct an infinite *Expr* value by writing a recursive Haskell program, we avoid it because it would not correspond to any expression of our language.) We also define the types *Var Real* and *Var Bool* for variable names.

```
data Var a where
  Real :: String → Var Real
  Bool :: String → Var Bool
```

But for brevity, we elide the constructors Real and Bool applied to literal strings in examples.

To interpret expressions in our language as generative stories, we write a function *sample*, which takes an expression and an environment as input and returns an *IO* action. To express that the type of the expression matches the outcome of the action, let's take the convenient shortcut of defining *Real* as a type synonym for *Double*, so that the Haskell type *IO Real* makes sense. The code for *sample* is straightforward:

```
type Real = Double

sample :: Expr a → Env → IO a
sample StdRandom   _ = getStdRandom random
sample (Lit x)     _ = return (fromRational x)
sample (Var v)     ρ = return (lookupEnv ρ v)
sample (Let v e e′) ρ = do x ← sample e ρ
                           sample e′ (extendEnv v x ρ)
sample (Neg e)     ρ = liftM  negate (sample e ρ)
sample (Exp e)     ρ = liftM  exp    (sample e ρ)
sample (Log e)     ρ = liftM  log    (sample e ρ)
sample (Not e)     ρ = liftM  not    (sample e ρ)
sample (Add e1 e2) ρ = liftM2 (+)    (sample e1 ρ)
                                     (sample e2 ρ)
sample (Less e1 e2) ρ = liftM2 (<)   (sample e1 ρ)
                                     (sample e2 ρ)
sample (If e e1 e2) ρ = do b ← sample e ρ
                           sample (if b then e1 else e2) ρ
```

As is typical of an interpreter, this *sample* function uses a type *Env* of environments (mapping variable names to values), along with the functions *lookupEnv* and *extendEnv* for querying and extending environments. For concision, here we opt to represent environments as functions. All this code is standard:

```
type Env = ∀a.Var a → a

lookupEnv :: Env → Var a → a
lookupEnv ρ = ρ

emptyEnv :: Env
emptyEnv v = error "Unbound"

extendEnv :: Var a → a → Env → Env
extendEnv (Real v) x _ (Real v′) | v ≡ v′ = x
extendEnv (Bool v) x _ (Bool v′) | v ≡ v′ = x
extendEnv _        _ ρ v′                 = ρ v′
```

We can now run our programs to get random outcomes:

```
> sample (Add StdRandom StdRandom) emptyEnv
0.8422448686660571
> sample (Add StdRandom StdRandom) emptyEnv
1.25881932199967
> sample (Let "x" StdRandom (Add (Var "x") (Var "x")))
      emptyEnv
0.23258391029872305
> sample (Let "x" StdRandom (Add (Var "x") (Var "x")))
      emptyEnv
1.1712041724765878
```

Your outcomes may vary, of course.

For more of a bird's-eye view of the distributions, we can take many independent samples then make a histogram out of each distribution. Two such histograms are shown in Figure 2. As those histograms indicate, the generative story of

```
Add StdRandom StdRandom
```

is different from the generative story of

```
Let "x" StdRandom (Add (Var "x") (Var "x"))
```

even though both expressions have the type *Real*. The latter expression means to choose just one random real number uniformly between 0 and 1, then double it. In general, Let means to choose an outcome once then use it any number of times. Thus, this is a call-by-value language whose side effect is random choice.

Although this language is small, our development below is wholly compatible with adding more types (such as *Integer*, products, and sums), arithmetic operations (such as division and sine), and distributions (such as normal, gamma, and Poisson). Besides, this language already lets us express many distributions, such as the *exponential distribution*, which ranges over the positive reals:

```
exponential :: Expr Real
exponential = Neg (Log StdRandom)
```

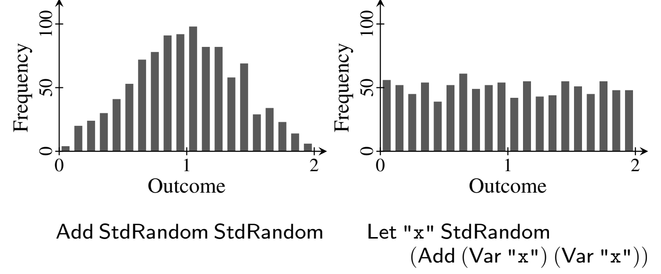

Figure 2. Histograms of two distributions over real numbers. Each histogram is produced by generating 1000 samples (as shown at the end of Section 2) and putting them into 20 equally spaced bins.

3. Composing Expectation Functionals

Although the *sample* interpreter is easy to write and intuitive to use, we shouldn't think that the *IO* action it returns is exactly the meaning of an expression. By "meaning" here, we mean what inference should preserve. We often want to reduce e to some other expression e' to speed up inference. The problem with treating *sample* e as the meaning of e is that such reduction would usually change the meaning of e, because *sample* e' makes different and fewer random choices than *sample* e.

For example, the expression Let "x" StdRandom (Lit 3) always produces the outcome 3, so we should be allowed to reduce it to just Lit 3, and an inference procedure shouldn't be obliged to consume any random seed before generating the 3. In other words, inference shouldn't be obliged to distinguish Lit 3 from Let "x" StdRandom (Lit 3), so we should assign these expressions the same meaning. They draw outcomes from the same distribution.

A less trivial example is that the definition of *sample* above specifies that, in an expression of the form Add e_1 e_2 or Less e_1 e_2, the random choices in e_1 must be made before the random choices in e_2, even though the order doesn't matter. Since addition is commutative, we should assign Add StdRandom (Neg StdRandom) and Add (Neg StdRandom) StdRandom the same meaning. They draw outcomes from the same distribution, even though sampling them starting from the same random seed gives different outcomes.

Thus, the meaning equivalence relation produced by the *sample* semantics is too fine-grained. To coarsen the equivalence properly, measure theory informs us to consider the *expected values* of distributions. Given an expression of type *Real*, its expected value, or *mean*, is basically what the average of many samples approaches as the number of samples approaches infinity. For example, if we run

$> \textit{sample}\ (\mathsf{Add\ StdRandom\ StdRandom})\ \textit{emptyEnv}$

many times and average the results, the average will approach 1 as we take more samples. In the examples above, the expressions Let "x" StdRandom (Lit 3) and Lit 3 both have the expected value 3, and the expressions Add e_1 e_2 and Add e_2 e_1 always have the same expected value.

3.1 The Expectation Interpreter

If the only question we ever ask about a distribution is "what is its expected value?", then it would be adequate for the meaning of each expression to equal its expected value. Unfortunately, there are other questions we ask whose answers differ on expressions with the same expected value. For example, given an expression of type *Real*, we might ask "what is the probability for its outcome to be less than $1/2$?"—perhaps to decide how to bet on it. The two distributions sampled in Figure 2 both have expected value 1, but the probability of being less than $1/2$ is $1/8$ for the first distribution and $1/4$ in the second distribution. Put differently, even though the two distributions have the same expected value, plugging them into the same *context*

$\mathsf{If}\ (\mathsf{Less}\ \ldots\ (\mathsf{Lit}\ (1/2)))\ (\mathsf{Lit}\ 1)\ (\mathsf{Lit}\ 0)$

gives two distributions with different expected values ($1/8 \neq 1/4$). Even if we know the expected value of an expression e, we don't necessarily know the expected value of the larger expression

$\mathsf{If}\ (\mathsf{Less}\ e\ (\mathsf{Lit}\ (1/2)))\ (\mathsf{Lit}\ 1)\ (\mathsf{Lit}\ 0)$

containing e.

Another way to phrase this complaint is to say that the expected-value interpretation is not compositional. That is, if we were to define a Haskell function

$\textit{mean} :: \textit{Expr Real} \to \textit{Env} \to \textit{Real}$

then it wouldn't be straightforward the way *sample* is. For example, as the counterexamples in Figure 2 show, there's no way to define

$\textit{mean}\ (\mathsf{If}\ (\mathsf{Less}\ e\ (\mathsf{Lit}\ (1/2)))\ (\mathsf{Lit}\ 1)\ (\mathsf{Lit}\ 0))\ \rho = \cdots$

in terms of *mean* e. In particular, it's incorrect to define

$$\begin{aligned}&\textit{mean}\ (\mathsf{If}\ (\mathsf{Less}\ e\ (\mathsf{Lit}\ (1/2)))\ (\mathsf{Lit}\ 1)\ (\mathsf{Lit}\ 0))\ \rho\\ &\quad = \textbf{if}\ \textit{mean}\ e < 1/2\ \textbf{then}\ 1\ \textbf{else}\ 0\end{aligned}$$

because it would give the result 0 on the distributions in Figure 2, not $1/8$ and $1/4$ as it should! People building a compiler for distributions, including the present authors, want compositionality in order to achieve separate compilation.

To make *mean* compositional, we add an argument to it to represent the context (Hughes 1995; Hinze 2000) that an expression is plugged into before its expected value is observed. The new function *expect* has the type signature

$\textit{expect} :: \textit{Expr}\ a \to \textit{Env} \to (a \to \textit{Real}) \to \textit{Real}$

where the third argument may or may not be the identity function. In other words, the question that *expect* $e\ \rho\ c$ asks is "what is the expected value of the distribution e in the environment ρ after its outcomes are transformed by the function c?". This value is also called the *expectation* of c with respect to the distribution. (To keep our derivation mathematically rigorous, we maintain the invariant that c is always measurable and non-negative (Pollard 2001, §2.4), but don't worry if you are not familiar with these side conditions.)

This generalization of *mean* is compositional: we can define *expect* on an expression in terms of *expect* on its subexpressions. Here is the definition:

$$\begin{aligned}
&\textit{expect}\ \mathsf{StdRandom}\ _\ c = \textstyle\int_0^1 \lambda x.\, c\ x\\
&\textit{expect}\ (\mathsf{Lit}\ x)\quad _\ c = c\ (\textit{fromRational}\ x)\\
&\textit{expect}\ (\mathsf{Var}\ v)\quad \rho\ c = c\ (\textit{lookupEnv}\ \rho\ v)\\
&\textit{expect}\ (\mathsf{Let}\ v\ e\ e')\ \rho\ c = \textit{expect}\ e\ \rho\ (\lambda x.\\
&\qquad\qquad \textit{expect}\ e'\ (\textit{extendEnv}\ v\ x\ \rho)\ c)\\
&\textit{expect}\ (\mathsf{Neg}\ e)\quad \rho\ c = \textit{expect}\ e\ \rho\ (\lambda x.\, c\ (\textit{negate}\ x))\\
&\textit{expect}\ (\mathsf{Exp}\ e)\quad \rho\ c = \textit{expect}\ e\ \rho\ (\lambda x.\, c\ (\textit{exp}\ \ \ x))\\
&\textit{expect}\ (\mathsf{Log}\ e)\quad \rho\ c = \textit{expect}\ e\ \rho\ (\lambda x.\, c\ (\textit{log}\ \ \ x))\\
&\textit{expect}\ (\mathsf{Not}\ e)\quad \rho\ c = \textit{expect}\ e\ \rho\ (\lambda x.\, c\ (\textit{not}\ \ \ x))\\
&\textit{expect}\ (\mathsf{Add}\ e_1\ e_2)\ \rho\ c = \textit{expect}\ e_1\ \rho\ (\lambda x.\\
&\qquad\qquad \textit{expect}\ e_2\ \rho\ (\lambda y.\, c\ (x+y)))\\
&\textit{expect}\ (\mathsf{Less}\ e_1\ e_2)\ \rho\ c = \textit{expect}\ e_1\ \rho\ (\lambda x.\\
&\qquad\qquad \textit{expect}\ e_2\ \rho\ (\lambda y.\, c\ (x<y)))\\
&\textit{expect}\ (\mathsf{If}\ e\ e_1\ e_2)\ \rho\ c = \textit{expect}\ e\ \rho\ (\lambda b.\\
&\qquad\qquad \textit{expect}\ (\textbf{if}\ b\ \textbf{then}\ e_1\ \textbf{else}\ e_2)\ \rho\ c)
\end{aligned}$$

3.2 Integrals Denoted by Random Choices

To understand this definition, let's start at the top.

The expected value of StdRandom is $1/2$, but that's not the only question that *expect* StdRandom needs to answer. Given any function c from reals to reals (and any environment ρ), *expect* StdRandom $\rho\ c$ is supposed to be the expected value of choosing a random number uniformly between 0 and 1 then applying c to it. That expected value is the integral of c from 0 to 1, which in conventional mathematical notation is written as $\int_0^1 c(x)\,dx$. (More precisely, we mean the Lebesgue integral of c with respect to the Lebesgue measure from 0 to 1.) In this paper, we blend Haskell and mathematical notation by writing this integral as $\int_0^1 \lambda x.\, c\ x$, as if there's a function

$\int_{\cdot}^{\cdot} \cdot :: \textit{Real} \to \textit{Real} \to (\textit{Real} \to \textit{Real}) \to \textit{Real}$

already defined. If we want to actually implement such a function, it could perform numerical integration. Or it could perform symbolic integration, or just print formulas containing $\int$, if we redefine the

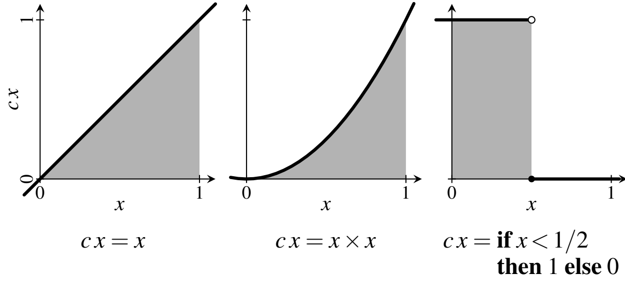

Figure 3. The expectation of 3 different functions with respect to the same distribution StdRandom, defined in terms of integration from 0 to 1 (the shaded areas)

Real type and overload the *Num* class to produce text or syntax trees. (We also notate multiplication by $\times$, so $c\ x$ always means applying c to x, as in Haskell, and not multiplying c by x.)

So for example, the expected value of *squaring* a uniform random number between 0 and 1 is

$$\begin{aligned} & \mathit{expect}\ \mathsf{StdRandom}\ \mathit{emptyEnv}\ (\lambda x. x\times x) \\ =\ & \textstyle\int_0^1 \lambda x. x\times x \\ =\ & 1/3 \end{aligned}$$

which is less than $1/2$ because squaring a number between 0 and 1 makes it smaller. And the probability that a uniform random number between 0 and 1 is less than $1/2$ is

$$\begin{aligned} & \mathit{expect}\ \mathsf{StdRandom}\ \mathit{emptyEnv}\ (\lambda x. \mathbf{if}\ x<1/2\ \mathbf{then}\ 1\ \mathbf{else}\ 0) \\ =\ & \textstyle\int_0^1 \lambda x. \mathbf{if}\ x<1/2\ \mathbf{then}\ 1\ \mathbf{else}\ 0 \\ =\ & 1/2 \end{aligned}$$

Figure 3 depicts these integrals.

The rest of the definition of *expect* is in *continuation-passing style* (Fischer 1993; Reynolds 1972; Strachey and Wadsworth 1974). The continuation c is the function whose expectation we want. The Lit and Var cases are deterministic (that is, they don't make any random choices), so the expectation of c with respect to those distributions simply applies c to one value. The unary operators (Neg, Exp, Log, Not) each compose the continuation with a mathematical function.

The remaining cases of *expect* involve multiple subexpressions and produce nested integrals if these subexpressions each yield integrals. For example, it follows from the definition that

$$\begin{aligned} & \mathit{expect}\ (\mathsf{Add}\ \mathsf{StdRandom}\ (\mathsf{Neg}\ \mathsf{StdRandom}))\ \mathit{emptyEnv}\ c \\ =\ & \mathit{expect}\ \mathsf{StdRandom}\ \mathit{emptyEnv}\ (\lambda x. \\ & \mathit{expect}\ \mathsf{StdRandom}\ \mathit{emptyEnv}\ (\lambda y. c\ (x+\mathit{negate}\ y))) \\ =\ & \textstyle\int_0^1 \lambda x. \int_0^1 \lambda y. c\ (x-y) \end{aligned}$$

The nesting order on the last line doesn't matter, as Tonelli's theorem (Pollard 2001, §4.4) assures it's equal to $\int_0^1 \lambda y. \int_0^1 \lambda x. c\ (x-y)$. In general, Tonelli's theorem lets us exchange nested integrals as long as the integrand (here $\lambda x\ y. c\ (x-y)$) is measurable and non-negative. Thus we could just as well define equivalently

$$\begin{aligned} \mathit{expect}\ (\mathsf{Add}\ e_1\ e_2)\ \rho\ c &= \mathit{expect}\ e_2\ \rho\ (\lambda y. \\ &\qquad \mathit{expect}\ e_1\ \rho\ (\lambda x. c\ (x+y))) \\ \mathit{expect}\ (\mathsf{Less}\ e_1\ e_2)\ \rho\ c &= \mathit{expect}\ e_2\ \rho\ (\lambda y. \\ &\qquad \mathit{expect}\ e_1\ \rho\ (\lambda x. c\ (x<y))) \end{aligned}$$

Besides compositionality, another benefit of generalizing *expect* is that it subsumes every question we can ask about a distribution. After all, a distribution is completely determined by the probability it assigns to each set of outcomes, and the probability of a set is the expectation of the set's characteristic function—the function that maps outcomes in the set to 1 and outcomes not in the set to 0. For example, if e is a closed expression of type *Real*, then the probability that the outcome of e is less than $1/2$ is

$$\mathit{expect}\ e\ \mathit{emptyEnv}\ (\lambda x. \mathbf{if}\ x<1/2\ \mathbf{then}\ 1\ \mathbf{else}\ 0)$$

Not only can we express the expected value of the distribution e as

$$\mathit{mean}\ e\ \rho = \mathit{expect}\ e\ \rho\ \mathit{id}$$

but we can also express all the other *moments* of e, such as the *variance* (the expected squared difference from the mean):

$$\begin{aligned} & \mathit{variance} :: \mathit{Expr}\ \mathit{Real} \to \mathit{Env} \to \mathit{Real} \\ & \mathit{variance}\ e\ \rho = \mathit{expect}\ e\ \rho\ (\lambda x. (x-\mathit{mean}\ e\ \rho)^2) \end{aligned}$$

To take another example, the ideal height of each histogram bar in Figure 2 is

$$\mathit{expect}\ e\ \mathit{emptyEnv}\ (\lambda x. \mathbf{if}\ \mathit{lo}<x\leqslant \mathit{hi}\ \mathbf{then}\ 1000\ \mathbf{else}\ 0)$$

where *lo* and *hi* are the bounds of the bin and 1000 is the total number of samples. (We abbreviate $\mathit{lo}<x\wedge x\leqslant \mathit{hi}$ to $\mathit{lo}<x\leqslant \mathit{hi}$.)

Mathematically speaking, every distribution corresponds to a *functional*, which is a function—sort of a generalized integrator—that takes as argument another function, namely the integrand c. This correspondence is expressed by *expect*, and it's injective. (In fact, it's bijective between measures and "increasing linear functionals with the Monotone Convergence property" (Pollard 2001, page 27).) Therefore, if $\mathit{expect}\ e\ \rho$ and $\mathit{expect}\ e'\ \rho$ are equal (in other words, if $\mathit{expect}\ e\ \rho\ c$ and $\mathit{expect}\ e'\ \rho\ c$ are equal for all c), then e and e' are equivalent and we can feel free to reduce e to e'.

In short, we define the meaning of the expression e in the environment ρ to be the functional $\mathit{expect}\ e\ \rho$. Returning to Figure 2, it follows from this definition that the meaning of

Add StdRandom StdRandom

in the empty environment is

$$\begin{aligned} & \mathit{expect}\ (\mathsf{Add}\ \mathsf{StdRandom}\ \mathsf{StdRandom})\ \mathit{emptyEnv} \\ =\ & \lambda c. \textstyle\int_0^1 \lambda x. \int_0^1 \lambda y. c\ (x+y) \end{aligned}$$

and the meaning of

Let "x" StdRandom (Add (Var "x") (Var "x"))

in the empty environment is

$$\begin{aligned} & \mathit{expect}\ (\mathsf{Let}\ \texttt{"x"}\ \mathsf{StdRandom}\ (\mathsf{Add}\ (\mathsf{Var}\ \texttt{"x"})\ (\mathsf{Var}\ \texttt{"x"}))) \\ & \qquad \mathit{emptyEnv} \\ =\ & \lambda c. \textstyle\int_0^1 \lambda x. c\ (x+x) \end{aligned}$$

The two expressions are not equivalent, because the two functionals are not equal: applied to the function $\lambda x. \mathbf{if}\ x<1/2\ \mathbf{then}\ 1\ \mathbf{else}\ 0$, the first functional returns $1/8$ whereas the second functional returns $1/4$. In this way, *expect* defines the semantics of our distribution language. Therefore, it naturally enters our specification of a probability density calculator, which we present next.

4. Specifying Probability Densities

Some distributions enjoy the existence of a *density function*. If the distribution is over the type a, then the density function maps from a to reals. Density functions are very useful in probabilistic inference: they underpin many concepts and techniques, including *maximum-likelihood estimation*, *conditioning*, and *Monte Carlo sampling* (MacKay 1998; Tierney 1998). We illustrate these applications in Section 7 below.

Intuitively, a density function is the outline of a histogram as the bin size approaches zero. For example, the two distributions in Figure 2 have the respective density functions shown in Figure 4. The shapes in the two figures are similar, but the histograms are

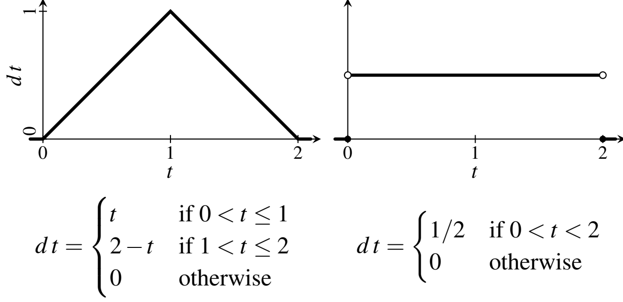

Figure 4. Density functions of the two distributions in Figure 2

randomly generated as this paper is typeset, whereas each density is a fixed mathematical function.

The precise definition of when a given function qualifies as a density for a given distribution depends on a *base* or *dominating measure*. When the distribution is over reals, the base measure is typically the Lebesgue measure over reals, in which case the definition amounts to the following.

Definition 1. A function $d :: Real \to Real$ is a *density* (with respect to the Lebesgue measure) for an expression $e :: Expr\ Real$ in an environment $\rho :: Env$ if and only if

$$expect\ e\ \rho\ c = \textstyle\int_{-\infty}^{\infty} \lambda t.\, d\ t \times c\ t$$

for all continuations $c :: Real \to Real$.

And when the distribution is over booleans, the base measure is typically the counting measure over booleans, in which case the definition amounts to the following.

Definition 2. A function $d :: Bool \to Real$ is a *density* (with respect to the counting measure) for an expression $e :: Expr\ Bool$ in an environment $\rho :: Env$ if and only if

$$expect\ e\ \rho\ c = sum\ [d\ t \times c\ t \mid t \leftarrow [True, False]]$$

for all continuations $c :: Bool \to Real$.

One way to gain intuition for these definitions is to let c be the characteristic function of a set. For example, in Definition 2, if

$$c = \lambda t.\, \mathbf{if}\ t\ \mathbf{then}\ 1\ \mathbf{else}\ 0$$

then $expect\ e\ \rho\ c$ is the probability of *True* according to e in ρ, and the equation requires $d\ True$ to equal it. In Definition 1, if

$$c = \lambda t.\, \mathbf{if}\ lo < t \leqslant hi\ \mathbf{then}\ 1\ \mathbf{else}\ 0$$

where lo and hi are the bounds of a histogram bin, then $expect\ e\ \rho\ c$ is the probability of that bin according to e in ρ, and the equation requires $\int_{lo}^{hi} d$ to equal the ideal proportion of that histogram bar.

These definitions use e and ρ just to represent the functional $expect\ e\ \rho$. Given e and ρ, because densities are useful, our goal is to find some function d that satisfies the specification above.

4.1 Examples of Densities and Their Non-existence

To illustrate these definitions, let's check that the functions in Figure 4 are indeed densities of their respective distributions. First let's consider the function on the right of Figure 4, which is supposed to be a density for

$$e = \mathsf{Let}\ \texttt{"x"}\ \mathsf{StdRandom}\ (\mathsf{Add}\ (\mathsf{Var}\ \texttt{"x"})\ (\mathsf{Var}\ \texttt{"x"}))$$

in *emptyEnv*. We start with the functional

$$m = expect\ e\ emptyEnv = \lambda c.\, \textstyle\int_0^1 \lambda x.\, c\ (x+x)$$

and reason equationally by univariate calculus. We extend the domain of integration from the interval $(0,1)$ to the entire real line:

$$m = \lambda c.\, \textstyle\int_{-\infty}^{\infty} \lambda x.\, (\mathbf{if}\ 0 < x < 1\ \mathbf{then}\ 1\ \mathbf{else}\ 0) \times c\ (x+x)$$

Then we change the integration variable from x to $t = x + x$:

$$m = \lambda c.\, \textstyle\int_{-\infty}^{\infty} \lambda t.\, (1/2) \times (\mathbf{if}\ 0 < t/2 < 1\ \mathbf{then}\ 1\ \mathbf{else}\ 0) \times c\ t$$

(The factor $1/2$ is the (absolute value of the) derivative of $x = t/2$ with respect to t.) Matching this equation against Definition 1 shows that

$$\begin{aligned} &\lambda t.\, (1/2) \times (\mathbf{if}\ 0 < t/2 < 1\ \mathbf{then}\ 1\ \mathbf{else}\ 0) \\ =\ &\lambda t.\, \mathbf{if}\ 0 < t < 2\ \mathbf{then}\ 1/2\ \mathbf{else}\ 0 \end{aligned}$$

is a density as desired. By the way, because changing the value of the integrand at a few points does not affect the integral, functions such as

$$\lambda t.\, \mathbf{if}\ t \equiv 1 \lor t \equiv 3\ \mathbf{then}\ 42\ \mathbf{else}\ \mathbf{if}\ 0 < t \leqslant 2\ \mathbf{then}\ 1/2\ \mathbf{else}\ 0$$

are densities just as well.

Turning to the function on the left of Figure 4, we want to check that it is a density for

$$e = \mathsf{Add}\ \mathsf{StdRandom}\ \mathsf{StdRandom}$$

in *emptyEnv*. Again we start with the functional

$$m = expect\ e\ emptyEnv = \lambda c.\, \textstyle\int_0^1 \lambda x.\, \int_0^1 \lambda y.\, c\ (x+y)$$

and do calculus. We extend the inner domain of integration from the interval $(0,1)$ to the entire real line:

$$m = \lambda c.\, \textstyle\int_0^1 \lambda x.\, \int_{-\infty}^{\infty} \lambda y.\, (\mathbf{if}\ 0 < y < 1\ \mathbf{then}\ 1\ \mathbf{else}\ 0) \times c\ (x+y)$$

Then we change the inner integration variable from y to $t = x + y$:

$$m = \lambda c.\, \textstyle\int_0^1 \lambda x.\, \int_{-\infty}^{\infty} \lambda t.\, (\mathbf{if}\ 0 < t - x < 1\ \mathbf{then}\ 1\ \mathbf{else}\ 0) \times c\ t$$

(No factor is required because the (partial) derivative of $y = t - x$ with respect to t is 1.) Tonelli's theorem lets us exchange the nested integrals:

$$m = \lambda c.\, \textstyle\int_{-\infty}^{\infty} \lambda t.\, \int_0^1 \lambda x.\, (\mathbf{if}\ 0 < t - x < 1\ \mathbf{then}\ 1\ \mathbf{else}\ 0) \times c\ t$$

Finally, because the inner integration variable x does not appear in the factor $c\ t$, we can pull $c\ t$ out (in other words, we can use the linearity of $\int_0^1 \cdot$):

$$m = \lambda c.\, \textstyle\int_{-\infty}^{\infty} \lambda t.\, (\int_0^1 \lambda x.\, \mathbf{if}\ 0 < t - x < 1\ \mathbf{then}\ 1\ \mathbf{else}\ 0) \times c\ t$$

Matching this last equation against Definition 1 shows that

$$\lambda t.\, \textstyle\int_0^1 \lambda x.\, \mathbf{if}\ 0 < t - x < 1\ \mathbf{then}\ 1\ \mathbf{else}\ 0$$

is a density. This formula can be further simplified to the desired closed form in the lower-left corner of Figure 4, either by hand or using a computer algebra system.

So far we have seen two examples of distributions and their densities. But not all distributions have a density. For example, when $e = \mathsf{Lit}\ 3$, we have

$$m = expect\ e\ \rho = \lambda c.\, c\ 3$$

so a density function d would have to satisfy

$$c\ 3 = \textstyle\int_{-\infty}^{\infty} \lambda t.\, d\ t \times c\ t$$

for all $c :: Real \to Real$. But if $c = \lambda t.\, \mathbf{if}\ t \equiv 3\ \mathbf{then}\ 1\ \mathbf{else}\ 0$, then the left-hand-side is 1 whereas the right-hand-side is

$$\textstyle\int_{-\infty}^{\infty} \lambda t.\, \mathbf{if}\ t \equiv 3\ \mathbf{then}\ d\ t\ \mathbf{else}\ 0$$

which is 0 no matter what $d :: Real \to Real$ is. So there's no such d.

It may surprise the reader that we say $\mathsf{Lit}\ 3$ has no density, because sometimes $\mathsf{Lit}\ 3$ is said to have a density, easily expressed in terms of the *Dirac delta function*. However, the Dirac delta is not a function in the ordinary sense but a *generalized function*, which only makes sense under integration. For example, the Dirac

delta doesn't map any number to anything, but rather integrates an ordinary function c to yield $c(0)$. Whereas an ordinary density function has the type $a \to Real$, a generalized function has the type $(a \to Real) \to Real$, same as produced by *expect*. So, generalized functions are essentially distributions, which is indeed what many people call them. In this paper we seek ordinary density functions, which are much more useful. For example, generalized functions cannot be multiplied together, which precludes their use in Monte Carlo sampling techniques such as *importance sampling* (illustrated in Section 7) and *Metropolis-Hastings sampling* (MacKay 1998; Tierney 1998).

4.2 Relation to Prior Density Calculators

Because density functions are useful, we want a program that automatically computes density functions from distribution expressions. Such a program can "compute functions" in two different senses of the phrase. Pfeffer's (2009, §5.2) density calculator is a random algorithm that produces a number. By running the algorithm many times and averaging the results, we can approximate the density of a distribution at a given point. In contrast, Bhat et al.'s (2012, 2013) density calculator deterministically produces an exact mathematical formula (which may contain integrals). As they suggest, we can then feed the formula to a computer algebra system or inference procedure to be analyzed or executed.

In the rest of this paper, we use equational reasoning to *derive* a compositional density calculator for the first time. Even though Definitions 1 and 2 seem to require conjuring a function out of thin air, the semantic specification above turns out to pave the way for the equational derivation below. The resulting calculator produces a density function that can be treated either as an exact formula (not necessarily closed form) or as an approximation algorithm, depending on whether $\int$ is treated as symbolic or numerical integration. It thus unites the density calculators of Pfeffer (2009) and Bhat et al. (2012, 2013). In particular, the variety of program constructs that those techniques and ours can handle appear to be the same.

5. Calculating Probability Densities

To recap, our goal in the rest of this paper to derive a program that, given e and ρ, finds a function d that satisfies Definitions 1 and 2.

We just saw that not every distribution has a density. Moreover, not every density can be represented using the operations on *Real* available to us. And even if a density can be represented, discovering it may require mathematical reasoning too advanced to be automated compositionally. For all these reasons, we have to relax our goal: let's write a program

$$density :: Expr\ a \to [Env \to a \to Real]$$

that maps each distribution expression e to a *list* of successes (Wadler 1985). For every element δ of the list, and for every environment ρ that binds all the free variables in e, we require that the function $\delta\ \rho$ be a density for e in ρ. That is, depending on if e has type *Expr Real* or *Expr Bool*, we want one of the two equations

$$expect\ e\ \rho\ c = \textstyle\int_{-\infty}^{\infty} \lambda t.\, \delta\ \rho\ t \times c\ t$$
$$expect\ e\ \rho\ c = sum\ [\delta\ \rho\ t \times c\ t \mid t \leftarrow [True, False]]$$

to hold for all δ, ρ, and c.

The list returned by *density* might be empty, but we'll do our best to keep it non-empty. For example, as shown in Section 4, we regret that *density* (Lit 3) must be the empty list, but

density (Let "x" StdRandom (Add (Var "x") (Var "x")))

and

density (Add StdRandom StdRandom)

can be the non-empty lists

$$[\lambda\rho\ t.\, \mathbf{if}\ 0 < t < 2\ \mathbf{then}\ 1/2\ \mathbf{else}\ 0]$$
$$[\lambda\rho\ t.\, \textstyle\int_0^1 \lambda x.\, \mathbf{if}\ 0 < t - x < 1\ \mathbf{then}\ 1\ \mathbf{else}\ 0]$$

respectively. Our density calculator ends up missing the first density (see Section 8.1), but it does find the second as soon as we introduce nondeterminism by concatenating lists (see Section 5.5).

The fact that not every distribution has a density holds another lesson for us. It turns out that *density* is not compositional. In other words, *density* on an expression cannot be defined in terms of *density* on its subexpressions, for the following reason. On one hand, Lit 3 and Lit 4 have no density, so *density* must map them both to the empty list. On the other hand, the larger expressions Add (Lit 3) StdRandom and Add (Lit 4) StdRandom have densities but different ones, so we want *density* to map them to different non-empty lists. Thus, *density* e does not determine *density* (Add e StdRandom). Instead, it will be in terms of *expect* e that we define *density* (Add e StdRandom). That is, although *density* is not compositional, the interpreter $\lambda e.\,(density\ e, expect\ e)$ is compositional (but see Section 8.2).

We define *density* e by structural induction on e.

5.1 Real Base Cases

An important base case is when $e =$ StdRandom: we define

$$density\ \mathsf{StdRandom} = [\lambda\rho\ t.\, \mathbf{if}\ 0 < t \wedge t < 1\ \mathbf{then}\ 1\ \mathbf{else}\ 0]$$

expressing the characteristic function of the unit interval. This clause satisfies Definition 1 because

$$\begin{aligned}
& expect\ \mathsf{StdRandom}\ \rho\ c \\
= &\quad \text{-- definition of } expect \\
& \textstyle\int_0^1 \lambda t.\, c\ t \\
= &\quad \text{-- extending the domain of integration} \\
& \textstyle\int_{-\infty}^{\infty} \lambda t.\, (\mathbf{if}\ 0 < t \wedge t < 1\ \mathbf{then}\ 1\ \mathbf{else}\ 0) \times c\ t
\end{aligned}$$

For the other base cases of type *Real*, we must fail, as discussed in Section 4.1.

$$\begin{aligned}
& density\ (\mathsf{Lit}\ _) &&= [\,] \\
& density\ (\mathsf{Var}\ (\mathsf{Real}\ _)) &&= [\,]
\end{aligned}$$

5.2 Boolean Cases

In a countable type such as *Bool* (in contrast to *Real*), every distribution has a density. In other words, there always exists a function d that satisfies Definition 2. We can derive it as follows:

$$\begin{aligned}
& expect\ e\ \rho\ c \\
= &\quad \text{-- } \eta\text{-expansion} \\
& expect\ e\ \rho\ (\lambda x.\, c\ x) \\
= &\quad \text{-- case analysis on } x \\
& expect\ e\ \rho\ (\lambda x.\, sum\ [\,(\mathbf{if}\ t \equiv x\ \mathbf{then}\ 1\ \mathbf{else}\ 0) \times c\ t \\
& \qquad\qquad\qquad\qquad \mid t \leftarrow [True, False]\,]) \\
= &\quad \text{-- Tonelli's theorem, or just linearity of } m \\
& sum\ [\,expect\ e\ \rho\ (\lambda x.\, \mathbf{if}\ t \equiv x\ \mathbf{then}\ 1\ \mathbf{else}\ 0) \times c\ t \\
& \qquad \mid t \leftarrow [True, False]\,]
\end{aligned}$$

In the right-hand-side above, $expect\ e\ \rho\ (\lambda x.\, \mathbf{if}\ t \equiv x\ \mathbf{then}\ 1\ \mathbf{else}\ 0)$ is just the probability of the boolean value t. Matching the right-hand-side against Definition 2 shows that the function $prob\ e\ \rho$ defined by

$$\begin{aligned}
& prob :: Expr\ Bool \to Env \to Bool \to Real \\
& prob\ e\ \rho\ t = expect\ e\ \rho\ (\lambda x.\, \mathbf{if}\ t \equiv x\ \mathbf{then}\ 1\ \mathbf{else}\ 0)
\end{aligned}$$

is a density for e in ρ. Accordingly, we define

$$\begin{aligned}
& density\ (\mathsf{Var}\ (\mathsf{Bool}\ v)) &&= [prob\ (\mathsf{Var}\ (\mathsf{Bool}\ v))] \\
& density\ (\mathsf{Not}\ e) &&= [prob\ (\mathsf{Not}\ e)] \\
& density\ (\mathsf{Less}\ e_1\ e_2) &&= [prob\ (\mathsf{Less}\ e_1\ e_2)]
\end{aligned}$$

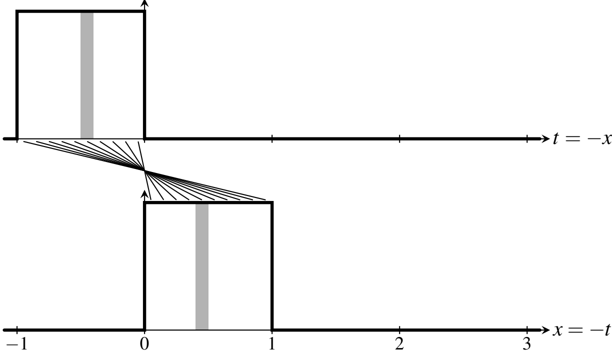

Figure 5. A density of Neg StdRandom (top) results from transforming a density of StdRandom (bottom)

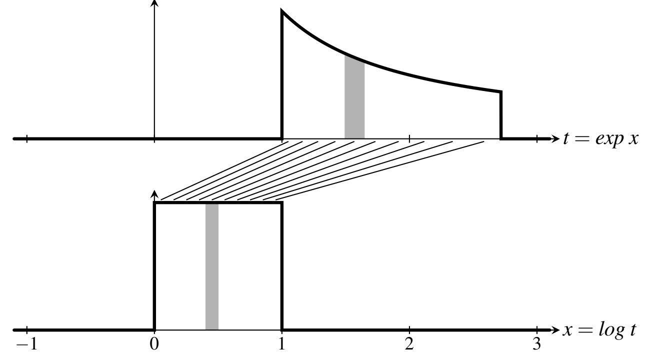

Figure 6. A density of Exp StdRandom (top) results from transforming a density of StdRandom (bottom)

5.3 Unary Cases

Things get more interesting in the other recursive cases. Take the case *density* (Neg e) for example. Suppose that the recursive call *density* e returns the successful result δ, so the induction hypothesis is that the equation

$$expect\ e\ \rho\ c = \int_{-\infty}^{\infty} \lambda t.\, \delta\ \rho\ t \times c\ t$$

holds for all ρ and c. We seek some δ' such that the equation

$$expect\ (\mathsf{Neg}\ e)\ \rho\ c = \int_{-\infty}^{\infty} \lambda t.\, \delta'\ \rho\ t \times c\ t$$

holds for all ρ and c. Starting with the left-hand-side, we calculate

$$\begin{aligned}
& expect\ (\mathsf{Neg}\ e)\ \rho\ c \\
= & \quad \text{-- definition of } expect \\
& expect\ e\ \rho\ (\lambda x.\, c\ (-x)) \\
= & \quad \text{-- induction hypothesis, substituting } \lambda x.\, c\ (-x) \text{ for } c \\
& \int_{-\infty}^{\infty} \lambda x.\, \delta\ \rho\ x \times c\ (-x) \\
= & \quad \text{-- changing the integration variable from } x \text{ to } t = -x \\
& \int_{-\infty}^{\infty} \lambda t.\, \delta\ \rho\ (-t) \times c\ t
\end{aligned}$$

Therefore, to match the goal, we define

$$density\ (\mathsf{Neg}\ e) = [\lambda \rho\ t.\, \delta\ \rho\ (-t) \mid \delta \leftarrow density\ e]$$

This clause says that the density of Neg e at t is the density of e at $-t$. This makes sense because the histogram of Neg e is the horizontal mirror image of the histogram of e. For example, Figure 5 depicts how the density of Neg StdRandom at t is the density of StdRandom at $-t$: when a sample from Neg StdRandom occurs between -0.5 and -0.4 (shaded above), it's because a sample from StdRandom occurred between 0.4 and 0.5 (shaded below).

A slightly more advanced case is *density* (Exp e). Again, we assume the induction hypothesis

$$expect\ e\ \rho\ c = \int_{-\infty}^{\infty} \lambda t.\, \delta\ \rho\ t \times c\ t$$

and seek some δ' satisfying

$$expect\ (\mathsf{Exp}\ e)\ \rho\ c = \int_{-\infty}^{\infty} \lambda t.\, \delta'\ \rho\ t \times c\ t$$

Starting with the left-hand-side, we calculate

$$\begin{aligned}
& expect\ (\mathsf{Exp}\ e)\ \rho\ c \\
= & \quad \text{-- definition of } expect \\
& expect\ e\ \rho\ (\lambda x.\, c\ (exp\ x)) \\
= & \quad \text{-- induction hypothesis, substituting } \lambda x.\, c\ (exp\ x) \text{ for } c \\
& \int_{-\infty}^{\infty} \lambda x.\, \delta\ \rho\ x \times c\ (exp\ x) \\
= & \quad \text{-- changing the integration variable from } x \text{ to } t = exp\ x \\
& \int_{0}^{\infty} \lambda t.\, (\delta\ \rho\ (log\ t)/t) \times c\ t \\
= & \quad \text{-- extending the domain of integration} \\
& \int_{-\infty}^{\infty} \lambda t.\, (\textbf{if}\ 0 < t\ \textbf{then}\ \delta\ \rho\ (log\ t)/t\ \textbf{else}\ 0) \times c\ t
\end{aligned}$$

Compared to the Neg case above, this calculation illustrates two complications:

1. The second-to-last step introduces the factor $1/t$, which is the (absolute value of the) derivative of $x = log\ t$ with respect to t. This factor makes sense because the histogram of Exp e is a *distorted* image of the histogram of e. For example, Figure 6 depicts how the density of Exp StdRandom at t is the density of StdRandom at $log\ t$, multiplied by $1/t$ because the interval $(e^{0.4}, e^{0.5})$ (shaded above) is *wider* than the interval $(0.4, 0.5)$ (shaded below) (Freedman et al. 2007, Chapter 3). After all, when a sample from Exp StdRandom occurs between $e^{0.4}$ and $e^{0.5}$, it's because a sample from StdRandom occurred between 0.4 and 0.5, so the two shaded areas in Figure 6 should be equal.
2. The last step introduces a conditional to account for the fact that the result of exponentiation is never negative.

In the end, to match the goal, we define

$$\begin{aligned}
density\ (\mathsf{Exp}\ e) = [&\lambda \rho\ t.\, \textbf{if}\ 0 < t\ \textbf{then}\ \delta\ \rho\ (log\ t)/t\ \textbf{else}\ 0 \\
&\mid \delta \leftarrow density\ e]
\end{aligned}$$

The case *density* (Log e) can be handled similarly, so we omit the derivation:

$$density\ (\mathsf{Log}\ e) = [\lambda \rho\ t.\, \delta\ \rho\ (exp\ t) \times exp\ t \mid \delta \leftarrow density\ e]$$

We can add other unary operators as well, such as reciprocal. (Alternatively, we can express reciprocal in terms of Exp, Neg, and Log, like with a slide rule.)

All these unary operators have in common that their densities *invert* their usual interpretation as functions: the density of Exp e at t uses the density of e at $log\ t$; the density of Neg e at t uses the density of e at $-t$; and so on. This pattern underlies the intuition that density calculation is backward sampling.

5.4 Conditional

For the case *density* (If $e\ e_1\ e_2$), suppose that the recursive calls *density* e_1 and *density* e_2 return the successful results δ_1 and δ_2. (It turns out that we don't need a density for the subexpression e.) We seek some δ' such that the equation

$$expect\ (\mathsf{If}\ e\ e_1\ e_2)\ \rho\ c = \int \lambda t.\, \delta'\ \rho\ t \times c\ t$$

holds for all ρ and c. Here the linear functional $\int \cdot$ is either $\int_{-\infty}^{\infty} \cdot$ (if e_1 and e_2 have type *Expr Real*) or $\lambda c.\, sum\ (map\ c\ [True, False])$ (if e_1 and e_2 have type *Expr Bool*). Starting with the left-hand-side, we calculate

$$\begin{aligned}
& \quad \textit{expect}\ (\mathsf{If}\ e\ e_1\ e_2)\ \rho\ c \\
= & \quad \text{-- definition of } \textit{expect} \\
& \quad \textit{expect}\ e\ \rho\ (\lambda b.\, \textit{expect}\ (\textbf{if}\ b\ \textbf{then}\ e_1\ \textbf{else}\ e_2)\ \rho\ c) \\
= & \quad \text{-- induction hypotheses} \\
& \quad \textit{expect}\ e\ \rho\ (\lambda b.\, \textstyle\int \lambda t.\, (\textbf{if}\ b\ \textbf{then}\ \delta_1\ \textbf{else}\ \delta_2)\ \rho\ t \times c\ t) \\
= & \quad \text{-- Tonelli's theorem, exchanging the integrals} \\
& \quad \text{-- } \textit{expect}\ e\ \rho\ (\lambda b.\dots)\ \text{and}\ \textstyle\int \lambda t.\dots \times c\ t \\
& \quad \textstyle\int \lambda t.\, \textit{expect}\ e\ \rho\ (\lambda b.\, (\textbf{if}\ b\ \textbf{then}\ \delta_1\ \textbf{else}\ \delta_2)\ \rho\ t) \times c\ t
\end{aligned}$$

Therefore, to match the goal, we define

$$\begin{aligned}
\textit{density}\ (\mathsf{If}\ e\ e_1\ e_2) = [\, & \lambda \rho\ t.\, \textit{expect}\ e\ \rho\ (\lambda b. \\
& \quad (\textbf{if}\ b\ \textbf{then}\ \delta_1\ \textbf{else}\ \delta_2)\ \rho\ t) \\
& \mid \delta_1 \leftarrow \textit{density}\ e_1, \delta_2 \leftarrow \textit{density}\ e_2\,]
\end{aligned}$$

Using the fact that $\lambda b.\dots$ above is just a function from *Bool* to *Real* (essentially the *Real* pair $(\delta_1\ \rho\ t, \delta_2\ \rho\ t)$), we can rewrite this definition more intuitively:

$$\begin{aligned}
\textit{density}\ (\mathsf{If}\ e\ e_1\ e_2) = [\, & \lambda \rho\ t.\, \textit{prob}\ e\ \rho\ \textit{True}\ \times \delta_1\ \rho\ t \\
& \quad +\ \textit{prob}\ e\ \rho\ \textit{False} \times \delta_2\ \rho\ t \\
& \mid \delta_1 \leftarrow \textit{density}\ e_1, \delta_2 \leftarrow \textit{density}\ e_2\,]
\end{aligned}$$

For example, if e is $\mathsf{Less\ StdRandom\ (Lit\ (1/2))}$, then

$$\textit{prob}\ e\ \rho\ \textit{True} = \textit{prob}\ e\ \rho\ \textit{False} = 1/2$$

and to sample from $\mathsf{If}\ e\ e_1\ e_2$ is to flip a fair coin to decide whether to sample from e_1 or from e_2. Accordingly, the density of $\mathsf{If}\ e\ e_1\ e_2$ is just the average of the densities of e_1 and e_2.

5.5 Binary Operators

Recall from Section 5.3 that the density of a unary operator $f(x)$ inverts f as a function of its operand x. The density of a binary operator $f(x,y)$ can invert f as a function of either x or y, treating the other operand as fixed. This choice brings a new twist to our derivation, namely that our density calculator can be *nondeterministic*: it can try multiple strategies for finding a density. If multiple strategies succeed, the resulting density functions are equivalent, in that they disagree only on a zero-probability set of outcomes. (But their subsequent performance may differ, so we keep them all.)

Take $\mathsf{Add}\ e_1\ e_2$ for example. The distribution denoted by $\mathsf{Add}\ e_1\ e_2$ is the convolution of the distributions denoted by e_1 and e_2. What we seek is some δ' such that the equation

$$\textit{expect}\ (\mathsf{Add}\ e_1\ e_2)\ \rho\ c = \textstyle\int_{-\infty}^{\infty} \lambda t.\, \delta'\ \rho\ t \times c\ t$$

holds for all ρ and c.

Again starting with the left-hand-side, we calculate

$$\begin{aligned}
& \quad \textit{expect}\ (\mathsf{Add}\ e_1\ e_2)\ \rho\ c \\
= & \quad \text{-- definition of } \textit{expect} \\
& \quad \textit{expect}\ e_1\ \rho\ (\lambda x.\, \textit{expect}\ e_2\ \rho\ (\lambda y.\, c\ (x+y)))
\end{aligned}$$

If the recursive call *density* e_2 returns the successful result δ_2, then the induction hypothesis lets us continue calculating as follows:

$$\begin{aligned}
= & \quad \text{-- induction hypothesis} \\
& \quad \textit{expect}\ e_1\ \rho\ (\lambda x.\, \textstyle\int_{-\infty}^{\infty} \lambda y.\, \delta_2\ \rho\ y \times c\ (x+y)) \\
= & \quad \text{-- changing the integration variable from } y \text{ to } t = x+y \\
& \quad \textit{expect}\ e_1\ \rho\ (\lambda x.\, \textstyle\int_{-\infty}^{\infty} \lambda t.\, \delta_2\ \rho\ (t-x) \times c\ t) \\
= & \quad \text{-- Tonelli's theorem} \\
& \quad \textstyle\int_{-\infty}^{\infty} \lambda t.\, \textit{expect}\ e_1\ \rho\ (\lambda x.\, \delta_2\ \rho\ (t-x)) \times c\ t
\end{aligned}$$

Therefore, having solved for y in $t = x+y$, we can define

$$\begin{aligned}
\textit{density}\ (\mathsf{Add}\ e_1\ e_2) = [\, & \lambda \rho\ t.\, \textit{expect}\ e_1\ \rho\ (\lambda x.\, \delta_2\ \rho\ (t-x)) \\
& \mid \delta_2 \leftarrow \textit{density}\ e_2\,]
\end{aligned}$$

For example, when e_1 is $\mathsf{Lit\ 3}$ and e_2 is $\mathsf{StdRandom}$, this definition amounts to solving for y in $t = 3+y$. The density calculator finds $y = t-3$ and thus returns

$$[\lambda \rho\ t.\, \textbf{if}\ 0 < t-3 < 1\ \textbf{then}\ 1\ \textbf{else}\ 0]$$

expressing the characteristic function of the interval $(3,4)$. We can also handle the example on the left of Figure 4 now: when e_1 and e_2 are both $\mathsf{StdRandom}$, the density calculator returns

$$[\lambda \rho\ t.\, \textstyle\int_0^1 \lambda x.\, \textbf{if}\ 0 < t-x < 1\ \textbf{then}\ 1\ \textbf{else}\ 0]$$

because *expect* $e_1\ \rho$ produces the integral and $\delta_2\ \rho\ (t-x)$ produces the conditional.

By analogous reasoning, we can also solve for x and define

$$\begin{aligned}
\textit{density}\ (\mathsf{Add}\ e_1\ e_2) = [\, & \lambda \rho\ t.\, \textit{expect}\ e_2\ \rho\ (\lambda y.\, \delta_1\ \rho\ (t-y)) \\
& \mid \delta_1 \leftarrow \textit{density}\ e_1\,]
\end{aligned}$$

Although solving for y and for x can produce overlapping lists (like when e_1 and e_2 are both $\mathsf{StdRandom}$), the two lists do not subsume each other. For example, because $\mathsf{Lit\ 3}$ has no density, only the first definition handles $\mathsf{Add\ (Lit\ 3)\ StdRandom}$ and only the second definition handles $\mathsf{Add\ StdRandom\ (Lit\ 3)}$. In the end, we define

$$\begin{aligned}
\textit{density}\ (\mathsf{Add}\ e_1\ e_2) = \ & [\lambda \rho\ t.\, \textit{expect}\ e_1\ \rho\ (\lambda x.\, \delta_2\ \rho\ (t-x)) \\
& \ \mid \delta_2 \leftarrow \textit{density}\ e_2\,] \\
& {+\!\!+}\ [\lambda \rho\ t.\, \textit{expect}\ e_2\ \rho\ (\lambda y.\, \delta_1\ \rho\ (t-y)) \\
& \ \mid \delta_1 \leftarrow \textit{density}\ e_1\,]
\end{aligned}$$

We can add other binary operators, such as multiplication, to our language and handle them similarly. (Alternatively, we can express multiplication in terms of Exp, Add, and Log, like with a slide rule.)

5.6 Variable Binding and Sharing

As with Add, an expression $\mathsf{Let}\ v\ e\ e'$ may have a density even if one of its subexpressions e and e' doesn't. We call v the bound variable, e the *right-hand-side* (Landin 1964; Peyton Jones 2003), and e' the *body* of the Let. There are two strategies for handling Let.

First, if the body e' has a density δ', then a density of the Let is the expectation of δ' with respect to the right-hand-side e. That is, if the recursive call *density* e' returns the successful result δ', then we calculate

$$\begin{aligned}
& \quad \textit{expect}\ (\mathsf{Let}\ v\ e\ e')\ \rho\ c \\
= & \quad \text{-- definition of } \textit{expect} \\
& \quad \textit{expect}\ e\ \rho\ (\lambda x.\, \textit{expect}\ e'\ (\textit{extendEnv}\ v\ x\ \rho)\ c) \\
= & \quad \text{-- induction hypothesis} \\
& \quad \textit{expect}\ e\ \rho\ (\lambda x.\, \textstyle\int \lambda t.\, \delta'\ (\textit{extendEnv}\ v\ x\ \rho)\ t \times c\ t) \\
= & \quad \text{-- Tonelli's theorem} \\
& \quad \textstyle\int \lambda t.\, (\textit{expect}\ e\ \rho\ (\lambda x.\, \delta'\ (\textit{extendEnv}\ v\ x\ \rho)\ t)) \times c\ t
\end{aligned}$$

Therefore, we can define

$$\begin{aligned}
& \textit{density}\ (\mathsf{Let}\ v\ e\ e') \\
& \quad = [\lambda \rho\ t.\, \textit{expect}\ e\ \rho\ (\lambda x.\, \delta'\ (\textit{extendEnv}\ v\ x\ \rho)\ t) \\
& \qquad \mid \delta' \leftarrow \textit{density}\ e'\,]
\end{aligned}$$

This strategy handles Let expressions that use the bound variable as a parameter. The right-hand-side can be deterministic, as in

```
Let "x" (Lit 3)
    (Add (Add (Var "x") (Var "x")) StdRandom)
```

or random, as in

```
Let "x" StdRandom
    (Add (Add (Var "x") (Var "x")) StdRandom)
```

These examples show that `Var "x"` can be used multiple times. That is, the outcome of the right-hand-side can be *shared*. We need Let in the language to introduce such sharing. However, this strategy fails on Let expressions whose bodies are deterministic, such as

```
Let "x" (Neg StdRandom)
    (Exp (Var "x"))
```

These Let expressions have densities only because their right-hand-sides are random. Hence we introduce another strategy for handling Let: check if the body of the Let uses the bound variable at most once. If so, we can inline the right-hand-side into the body. That is, we can replace Let $v\ e\ e'$ by the result of substituting e for v in e', which we write as $e'\{v \mapsto e\}$. (This substitution operation sometimes needs to rename variables in e' to avoid capture.) This replacement preserves the meaning of the Let expression even if the body is random. For example, we can handle the expression

$$e_1 = \mathsf{Let}\ \texttt{"x"}\ (\mathsf{Neg\ StdRandom})\\ \quad (\mathsf{Add\ StdRandom}\ (\mathsf{Exp}\ (\mathsf{Var}\ \texttt{"x"})))$$

by turning it into the equivalent expression

$$e_2 = \mathsf{Add\ StdRandom}\ (\mathsf{Exp}\ (\mathsf{Neg\ StdRandom}))$$

To see this equivalence, apply the definition of *expect* to e_1 and e_2:

$$\mathit{expect}\ e_1\ \rho\ c = \textstyle\int_0^1 \lambda x.\int_0^1 \lambda t.\,c\ (t + \mathit{exp}\ (-x))$$
$$\mathit{expect}\ e_2\ \rho\ c = \textstyle\int_0^1 \lambda t.\int_0^1 \lambda x.\,c\ (t + \mathit{exp}\ (-x))$$

Then use Tonelli's theorem to move inward the outer integral $\int_0^1 \lambda x$ in $\mathit{expect}\ e_1\ \rho\ c$, which corresponds to the random choice made in Neg StdRandom. If we think of random choice as a side effect, then Tonelli's theorem lets us delay evaluating the right-hand-side Neg StdRandom until the body Add StdRandom (Exp (Var "x")) actually uses the bound variable "x".

In general, Tonelli's theorem tells us that delayed evaluation preserves the expectation semantics of the expression Let $v\ e\ e'$ when the body e' uses the bound variable v exactly once. Moreover, in the case where e' never uses v, delayed evaluation also preserves the expectation semantics, but for a different reason. If e' never uses v, then $\mathit{expect}\ e'\ (\mathit{extendEnv}\ v\ x\ \rho)\ c = \mathit{expect}\ e'\ \rho\ c$, so

$$\begin{aligned}
&\mathit{expect}\ (\mathsf{Let}\ v\ e\ e')\ \rho\ c\\
=\ &\quad \text{-- definition of } \mathit{expect}\\
&\mathit{expect}\ e\ \rho\ (\lambda x.\,\mathit{expect}\ e'\ (\mathit{extendEnv}\ v\ x\ \rho)\ c)\\
=\ &\quad \text{-- } e' \text{ never uses } v\\
&\mathit{expect}\ e\ \rho\ (\lambda x.\,\mathit{expect}\ e'\ \rho\ c)\\
=\ &\quad \text{-- pull } \mathit{expect}\ e'\ \rho\ c \text{ out of the integral } \mathit{expect}\ e\ \rho\ (\lambda x.\dots)\\
&\mathit{expect}\ e\ \rho\ (\lambda x.\,1) \times \mathit{expect}\ e'\ \rho\ c
\end{aligned}$$

Finally, a simple induction on e shows that $\mathit{expect}\ e\ \rho\ (\lambda x.\,1)$ always equals 1 in our language. In other words, the distributions denoted in our language are all *probability* distributions.

Backed by this reasoning, we put the two strategies together to define

$$\begin{aligned}
&\mathit{density}\ (\mathsf{Let}\ v\ e\ e')\\
&\quad = [\lambda\rho\ t.\,\mathit{expect}\ e\ \rho\ (\lambda x.\,\delta'\ (\mathit{extendEnv}\ v\ x\ \rho)\ t)\\
&\qquad\quad \mid \delta' \leftarrow \mathit{density}\ e'] \qquad \text{-- first strategy}\\
&\quad +\!\!+\ [\delta' \mid \mathit{usage}\ e'\ v \leqslant \mathit{AtMostOnce}\\
&\qquad\quad , \delta' \leftarrow \mathit{density}\ (e'\{v \mapsto e\})] \quad \text{-- second strategy}
\end{aligned}$$

The condition $\mathit{usage}\ e'\ v \leqslant \mathit{AtMostOnce}$ above tests conservatively whether the expression e' uses the variable v at most once—in other words, whether v is *not shared* in e'. This test serves the purpose of Bhat et al.'s (2012) *active variables* and independence test. We relegate its implementation to Appendix A.

6. Approximating Probability Densities

We are done deriving our density calculator! It produces output rife with integrals. The definition of *density* itself does not contain integrals, but *expect* StdRandom contains an integral, and *density* invokes *expect* in the boolean, If, Add, and Let cases. For example, Section 5.5 shows one success of our density calculator:

$$\begin{aligned}
&\mathit{density}\ (\mathsf{Add\ StdRandom\ StdRandom})\\
&\quad = [\lambda\rho\ t.\,\mathit{expect}\ \mathsf{StdRandom}\ \rho\ (\lambda x.\,\delta_2\ \rho\ (t-x))\\
&\qquad \mid \delta_2 \leftarrow \mathit{density}\ \mathsf{StdRandom}] \qquad +\!\!+ \cdots\\
&\quad = [\lambda\rho\ t.\textstyle\int_0^1 \lambda x.\,\textbf{if}\ 0 < t-x < 1\ \textbf{then}\ 1\ \textbf{else}\ 0] +\!\!+ \cdots
\end{aligned}$$

One way to use *density* is to feed its output to a symbolic integrator, as Bhat et al. (2012) suggest. If we're lucky, we might get a closed form that can be run as an exact deterministic algorithm. For example, Maxima, Maple, and Mathematica can each simplify the successful result above to the closed form in the lower-left corner of Figure 4. To produce output that those systems can parse, we would redefine the *Real* type and overload the *Num* class, which is not difficult to do.

Even without computer algebra or without eliminating all integrals, we can execute the density found as a *randomized* algorithm whose *expected* output is the density at the given point. All it takes is interpreting each call from *density* to *expect* as sampling randomly from a distribution. This interpretation is a form of numerical integration, carried out by Pfeffer's (2009, §5.2) approximate algorithm for density estimation. For example, we can interpret the successful result above, as is, as the following randomized (and embarrassingly parallel) algorithm:

> Given ρ and t, choose a random real number x uniformly between 0 and 1, then compute **if** $0 < t-x < 1$ **then** 1 **else** 0.

When time is about to run out, we average the results from repeated independent runs of this algorithm.

7. Applications of Our Density Calculator

Returning to our motivation, let us briefly demonstrate two ways to use density functions in probabilistic inference.

7.1 Computing and Comparing Likelihoods

Imagine that we are adjudicating between two competing scientific hypotheses, which we model by two generative stories e_1 and e_2. Using their density functions, we can update our belief in light of empirical evidence.

Concretely, suppose we conduct many independent trials of the following experiment:

1. We use a known device to draw a quantity from the *exponential* distribution (defined at the end of Section 2). We cannot observe this quantity directly, but it can cause effects that we can observe. So we give it a name x.
2. We use an unknown device to transform x to another quantity, which we do observe. The goal of the experiment is to find out what the unknown device does. Suppose we know that either the unknown device produces x as is, or it produces $e^x - 1$.

We can model our two hypotheses about the unknown device by two generative stories whose outcomes are possible observations:

$$\begin{aligned}
&e_1, e_2 :: \mathit{Expr\ Real}\\
&e_1 = \mathsf{Let}\ \texttt{"x"}\ \mathit{exponential}\ (\mathsf{Var}\ \texttt{"x"})\\
&e_2 = \mathsf{Let}\ \texttt{"x"}\ \mathit{exponential}\ (\mathsf{Add}\ (\mathsf{Exp}\ (\mathsf{Var}\ \texttt{"x"}))\ (\mathsf{Lit}\ (-1)))
\end{aligned}$$

Initially, say that we judge the two hypotheses to be equally probable. Then we start the experiment and the observations roll in:

$$\begin{aligned}
&\mathit{obs} :: [\mathit{Real}]\\
&\mathit{obs} = [3.07, 0.74, 2.23]
\end{aligned}$$

What do we believe now?

Our calculator finds the following densities for e_1 and e_2:

$$\begin{aligned}
&d_1, d_2 :: \mathit{Real} \to \mathit{Real}\\
&d_1\ t = (\textbf{if}\ 0 < \mathit{exp}\ (-t) \wedge \mathit{exp}\ (-t) < 1\ \textbf{then}\ 1\ \textbf{else}\ 0)\\
&\qquad \times \mathit{exp}\ (-t)
\end{aligned}$$

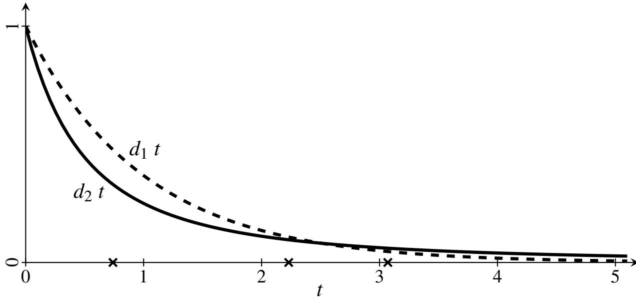

Figure 7. Likelihoods for two competing hypotheses. The crosses on the horizontal axis mark the *obs*erved outcomes.

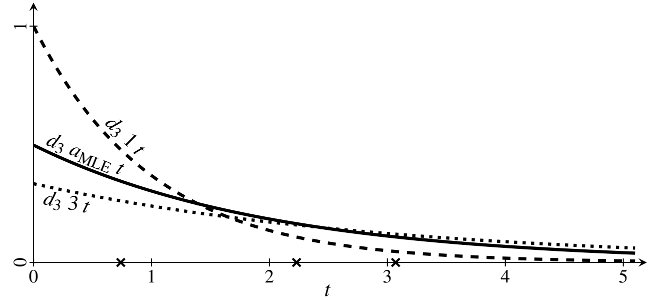

Figure 8. Likelihoods for three members of an infinite family d_3 of competing hypotheses. The solid line is the MLE likelihood. The crosses on the horizontal axis mark the *obs*erved outcomes.

```
d2 t = if 0 < t − (−1) then d1 (log (t − (−1)))/(t − (−1))
                       else 0
```

Figure 7 plots these densities. They are called *likelihoods* because they are densities of distributions over observations.

The likelihood $d_1\ t$ measures how likely it would be for us to observe t in a trial if the hypothesis e_1 were true. Moreover, because the trials are independent, the likelihood of multiple observations is just the product of their likelihoods. Thus *product* (*map* d_1 *obs*) measures how likely our *obs*ervations would be if the hypothesis e_1 were true, and similarly for e_2. To compare the hypotheses against each other, we calculate the ratio of the likelihoods:

```
> product (map d1 obs)/product (map d2 obs)
1.2461022752116167
```

The likelihood ratio is above 1, which means the evidence favors our first hypothesis—that is, the unknown device produces x as is. We see this faintly in Figure 7: above where the crosses mark the *obs*ervations, the value of d_1 tends to be greater than the value of d_2.

Whenever we have two or more hypotheses to choose from, the one with the greatest observation likelihood is called the *maximum-likelihood estimate* (MLE). So the MLE between e_1 and e_2 above is e_1. But we can also choose from an infinite family of hypotheses. In the experiment above for example, we can hypothesize instead that we observe $a \times x$ in each trial, where a is an unobserved positive parameter that describes the unknown device. We can model this infinite family of hypotheses by an expression with a free variable "a":

```
e3 :: Expr Real
e3 = Let "x" exponential (mul (Var "a") (Var "x"))
mul :: Expr Real → Expr Real → Expr Real
mul a x = Exp (Add (Log a) (Log x))    -- a > 0 ∧ x > 0
```

Our calculator finds a density for e_3 that simplifies algebraically to

```
d3 :: Real → Real → Real
d3 a = λt. if t > 0 then exp (−t/a)/a else 0
```

Figure 8 plots three members of this family of densities.

Using a bit of differential calculus, we can find the exact value of a that maximizes the likelihood *product* (*map* ($d_3\ a$) *obs*). That value is the MLE of the parameter a. Somewhat intuitively, it is the average of *obs*, represented by the solid line in Figure 8:

```
> let a_MLE = sum obs/fromIntegral (length obs)
> a_MLE
2.013333333333333
```

7.2 Importance Sampling

We can also use densities for *importance sampling* (MacKay 1998). Suppose we have a density function d (called the *target*), and we wish to sample repeatedly from the distribution with density d, perhaps to estimate the expectation of some function c with respect to the target distribution. Unfortunately, we don't know how to sample from the target distribution; in other words, we don't know any generative story with density d. We can pick a generative story e_1 we do know (called the *proposal*), find a density d_1 for e_1, and sample from e_1 repeatedly instead. To correct for the difference between the target and proposal densities, we pair each outcome t from e_1 with the *importance weight* $d\ t/d_1\ t$.

```
importance_sample :: (Real → Real) → IO (Real, Real)
importance_sample d = do t ← sample e1 emptyEnv
                         return (t, d t/d1 t)
```

In particular, to estimate the expectation of the function c with respect to the target distribution, we compute the *weighted* average of $c\ t$ where t is drawn from the proposal distribution.

```
estimate_expectation :: (Real → Real) → (Real → Real)
                                      → IO Real
estimate_expectation d c = do
  samples ← replicateM 10000 (importance_sample d)
  return (sum [c t × w | (t, w) ← samples]
         /sum [      w | (t, w) ← samples])
```

For example, suppose we don't know how to sample from the target distribution with density

```
d :: Real → Real
d t = exp (−t³)
```

but we would like to estimate the expectation of *sin* with respect to it. We can use the definitions above to estimate the expectation:

```
> estimate_expectation d sin
0.4508234334172205
```

8. Properties of Our Density Calculator

As explained at the beginning of Section 5, we want our density calculator to succeed as often as possible and to be compositional. Unfortunately, *density* does not succeed as often as we want. However, it can be made compositional.

8.1 Incompleteness

As shown in Section 4, the distribution

```
Let "x" StdRandom (Add (Var "x") (Var "x"))
```

has a density function. In particular, it would be correct if

```
density (Let "x" StdRandom (Add (Var "x") (Var "x")))
```

were to return the non-empty list

$$[\lambda\rho\ t.\ \textbf{if}\ 0<t<2\ \textbf{then}\ 1/2\ \textbf{else}\ 0]$$

Nevertheless, our *density* function returns the empty list, because

$$\begin{aligned}&\mathit{usage}\ (\mathsf{Add}\ (\mathsf{Var}\ \texttt{"x"})\ (\mathsf{Var}\ \texttt{"x"}))\ \texttt{"x"} = \mathit{Unknown}\\&\mathit{density}\ [\mathsf{Add}\ (\mathsf{Var}\ \texttt{"x"})\ (\mathsf{Var}\ \texttt{"x"})] \quad = []\end{aligned}$$

and our code does not know $x+x=2\times x$.

This example shows there is room for our code to improve by succeeding more often. Transformations that reduce the usage of variables (for example, rewriting $x+x$ to $2\times x$) would help, as would computer-algebra facilities for inverting a function that is expressed using non-invertible primitives (such as x^3+x). Unfortunately, those improvements would make it harder to keep a density calculator compositional and equationally derived.

Another source of incompleteness is demonstrated by the following example. Suppose e is some distribution expression that generates both positive and negative reals. For example, e could be $\mathsf{Add}\ \mathit{exponential}\ (\mathsf{Lit}\ (-42))$. We can express taking the absolute value of the outcome of e:

$$\begin{aligned}e' = \mathsf{Let}\ \texttt{"x"}\ e\ (&\mathsf{If}\ (\mathsf{Less}\ (\mathsf{Lit}\ 0)\ (\mathsf{Var}\ \texttt{"x"}))\\&\quad(\mathsf{Var}\ \texttt{"x"})\\&\quad(\mathsf{Neg}\ (\mathsf{Var}\ \texttt{"x"})))\end{aligned}$$

If e has a density d, then e' has a fairly intuitive density d':

$$\begin{aligned}&d' :: \mathit{Real} \to \mathit{Real}\\&d'\ t = \textbf{if}\ t>0\ \textbf{then}\ d\ t + d\ (-t)\ \textbf{else}\ 0\end{aligned}$$

But our calculator cannot find d', because the branches $\mathsf{Var}\ \texttt{"x"}$ and $\mathsf{Neg}\ (\mathsf{Var}\ \texttt{"x"})$ have no density separately from the Let.

To handle these cases, it turns out that we can generalize our density calculator so that, when applied to $\mathsf{Let}\ \texttt{"x"}\ e\ (\mathsf{Var}\ \texttt{"x"})$ or $\mathsf{Let}\ \texttt{"x"}\ e\ (\mathsf{Neg}\ (\mathsf{Var}\ \texttt{"x"}))$, it not only returns a density function but also *updates the environment*, mapping "x" to t or to $-t$ respectively. The condition $\mathsf{Less}\ (\mathsf{Lit}\ 0)\ (\mathsf{Var}\ \texttt{"x"})$ can then be evaluated in the updated environment. In other words, it helps to generalize the environment to the heap of a lazy evaluator (Launchbury 1993), and to delay evaluating the condition.

8.2 Compositionality

As promised above Section 5.1, our definition of *density e* necessarily uses not only the *density* of the subexpressions of e, but also *expect*. But to handle Let, we strayed even further from perfect compositionality: our definition depends on substitution and *usage*, two more functions defined by structural induction on expressions. Can we still express *density* as a special case of a compositional and more general function, just as *mean* is a special case of the compositional and more general function *expect*? The answer turns out to be yes—we just need to rearrange the code already derived above. This is good news for people building a compiler from distributions to densities, including the present authors, because compositionality enables separate compilation.

If we had only used *expect* and *usage* to define *density*, it would have been straightforward to generalize *density* to a compositional function: just specify the omnibus interpretation *generalDensity* by

$$\begin{aligned}&\textbf{data}\ \mathit{GeneralDensity}\ a = \mathit{GD}\ \{\\&\quad \mathit{gdExpect} :: \mathit{Env} \to (a \to \mathit{Real}) \to \mathit{Real},\\&\quad \mathit{gdUsage} :: \forall b.\mathit{Var}\ b \to \mathit{Usage},\\&\quad \mathit{gdDensity} :: [\mathit{Env} \to a \to \mathit{Real}]\}\\&\mathit{generalDensity} :: \mathit{Expr}\ a \to \mathit{GeneralDensity}\ a\\&\mathit{generalDensity}\ e = \mathit{GD}\ \{\mathit{gdExpect} = \mathit{expect}\ e,\\&\qquad\qquad\qquad\qquad\quad \mathit{gdUsage} = \mathit{usage}\ e,\\&\qquad\qquad\qquad\qquad\quad \mathit{gdDensity} = \mathit{density}\ e\}\end{aligned}$$

and fuse it with our clauses defining *expect*, *usage*, and *density*, so as to define *generalDensity e* purely by structural induction on e. For example, the new clause defining *generalDensity* on Add expressions would read

$$\begin{aligned}&\mathit{generalDensity}\ (\mathsf{Add}\ e_1\ e_2) = \mathit{GD}\ \{\\&\quad \mathit{gdExpect} = \lambda\rho\ c.\ \mathit{gdExpect}\ \mathit{gd}_1\ \rho\ (\lambda x.\\&\qquad\qquad\qquad \mathit{gdExpect}\ \mathit{gd}_2\ \rho\ (\lambda y.\ c\ (x+y))),\\&\quad \mathit{gdUsage} = \lambda v.\ \mathit{gdUsage}\ \mathit{gd}_1\ v \oplus \mathit{gdUsage}\ \mathit{gd}_2\ v,\\&\quad \mathit{gdDensity} = [\lambda\rho\ t.\ \mathit{gdExpect}\ \mathit{gd}_1\ \rho\ (\lambda x.\ \delta_2\ \rho\ (t-x))\\&\qquad\qquad \mid \delta_2 \leftarrow \mathit{gdDensity}\ \mathit{gd}_2]\\&\qquad\qquad +\!\!+\ [\lambda\rho\ t.\ \mathit{gdExpect}\ \mathit{gd}_2\ \rho\ (\lambda y.\ \delta_1\ \rho\ (t-y))\\&\qquad\qquad \mid \delta_1 \leftarrow \mathit{gdDensity}\ \mathit{gd}_1]\}\\&\quad \textbf{where}\ \mathit{gd}_1 = \mathit{generalDensity}\ e_1\\&\qquad\qquad \mathit{gd}_2 = \mathit{generalDensity}\ e_2\end{aligned}$$

collecting the definition of *expect* $(\mathsf{Add}\ e_1\ e_2)$ in Section 3.1, the definition of *usage* $(\mathsf{Add}\ e_1\ e_2)$ in Appendix A, and the definition of *density* $(\mathsf{Add}\ e_1\ e_2)$ in Section 5.5. This is the *tupling transformation* (Pettorossi 1984; Bird 1980) applied to our *dependent interpretations* (Gibbons and Wu 2014, §4.2).

Our use of *density* $(e'\{v \mapsto e\})$ to define *density* $(\mathsf{Let}\ v\ e\ e')$ in Section 5.6 complicates our quest for compositionality, because the recursive argument $e'\{v \mapsto e\}$ is not necessarily a subexpression of $\mathsf{Let}\ v\ e\ e'$. Instead of substituting e for v, we need the semantic analogue: some map, which we call *SEnv* for "static environment", that associates the variable v to the *expect* and *density* interpretations of e. We group these interpretations into a record type *General*. And instead of storing values in *Env* and renaming variables to avoid capture, we need the semantic analogue: storing values in lists, which we call *DEnv* for "dynamic environment", and allocating a fresh position in the lists for each variable. The type *General a* below contrasts with the type *GeneralDensity a* above.

$$\begin{aligned}&\textbf{data}\ \mathit{SEnv} = \mathit{SEnv}\ \{\\&\quad \mathit{freshReal}, \mathit{freshBool} :: \mathit{Int},\\&\quad \mathit{lookupSEnv} :: \forall a.\mathit{Var}\ a \to \mathit{General}\ a\}\\&\textbf{data}\ \mathit{General}\ a = \mathit{General}\ \{\\&\quad \mathit{gExpect} :: \mathit{DEnv} \to (a \to \mathit{Real}) \to \mathit{Real},\\&\quad \mathit{gDensity} :: [\mathit{DEnv} \to a \to \mathit{Real}]\}\\&\textbf{data}\ \mathit{DEnv} = \mathit{DEnv}\ \{\\&\quad \mathit{lookupReal} :: [\mathit{Real}],\\&\quad \mathit{lookupBool} :: [\mathit{Bool}]\}\end{aligned}$$

We call our omnibus interpretation *general*. It maps each distribution expression to its *usage* alongside a function from static environments to *expect* and *density* interpretations. The definition of *general* is mostly rearranging the code in Sections 3.1 and 5, so we relegate it to Appendix B.

$$\begin{aligned}&\mathit{general} :: \mathit{Expr}\ a \to (\forall b.\mathit{Var}\ b \to \mathit{Usage},\\&\qquad\qquad\qquad\qquad \mathit{SEnv} \to \mathit{General}\ a)\end{aligned}$$

At the top-level scope where the processing of a closed distribution expression commences, the static environment maps every variable name to an error and begins allocation at list position 0, matching the initially empty dynamic environment.

$$\begin{aligned}&\mathit{emptySEnv} :: \mathit{SEnv}\\&\mathit{emptySEnv} = \mathit{SEnv}\ \{\mathit{freshReal} = 0, \mathit{freshBool} = 0,\\&\qquad\qquad\qquad \mathit{lookupSEnv} = \lambda v.\ \mathit{error}\ \texttt{"Unbound"}\}\\&\mathit{emptyDEnv} :: \mathit{DEnv}\\&\mathit{emptyDEnv} = \mathit{DEnv}\ \{\mathit{lookupReal} = [], \mathit{lookupBool} = []\}\end{aligned}$$

We can finally define our density calculator as a special case of the compositional function *general*:

$$\begin{aligned}&\mathit{runDensity} :: \mathit{Expr}\ a \to [a \to \mathit{Real}]\\&\mathit{runDensity}\ e = [\delta\ \mathit{emptyDEnv}\\&\qquad\qquad \mid \delta \leftarrow \mathit{gDensity}\ (\mathit{snd}\ (\mathit{general}\ e)\ \mathit{emptySEnv})]\end{aligned}$$

9. Conclusion

We have turned a specification of density functions in terms of expectation functionals into a syntax-directed implementation that supports separate compilation. Our equational derivation draws from algebra, integral calculus, and λ-calculus. It suggests that program calculation and transformation are powerful ways to turn expressive probabilistic models into effective inference procedures. We are investigating this hypothesis in ongoing work.

Acknowledgments

Thanks to Jacques Carette, Praveen Narayanan, Norman Ramsey, Dylan Thurston, Mitchell Wand, Robert Zinkov, and anonymous reviewers for helpful comments and discussions.

This research was supported by DARPA grant FA8750-14-2-0007, NSF grant CNS-0723054, Lilly Endowment, Inc. (through its support for the Indiana University Pervasive Technology Institute), and the Indiana METACyt Initiative. The Indiana METACyt Initiative at IU is also supported in part by Lilly Endowment, Inc.

References

S. Bhat, A. Agarwal, R. Vuduc, and A. Gray. A type theory for probability density functions. In *Proceedings of POPL 2012*, pages 545–556. ACM Press, 2012.

S. Bhat, J. Borgström, A. D. Gordon, and C. V. Russo. Deriving probability density functions from probabilistic functional programs. In N. Piterman and S. A. Smolka, editors, *Proceedings of TACAS 2013*, number 7795 in Lecture Notes in Computer Science, pages 508–522. Springer, 2013.

R. S. Bird. Tabulation techniques for recursive programs. *ACM Computing Surveys*, 12(4):403–417, 1980.

K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of ICFP 2000*, pages 268–279. ACM Press, 2000.

M. J. Fischer. Lambda-calculus schemata. *Lisp and Symbolic Computation*, 6(3–4):259–288, 1993.

D. Freedman, R. Pisani, and R. Purves. *Statistics*. W. W. Norton, fourth edition, 2007.

J. Gibbons and N. Wu. Folding domain-specific languages: Deep and shallow embeddings (functional pearl). In J. Jeuring and M. M. T. Chakravarty, editors, *Proceedings of ICFP 2014*, pages 339–347. ACM Press, 2014.

R. Hinze. Deriving backtracking monad transformers. In *Proceedings of ICFP 2000*, pages 186–197. ACM Press, 2000.

J. Hughes. The design of a pretty-printing library. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming: 1st International Spring School on Advanced Functional Programming Techniques*, number 925 in Lecture Notes in Computer Science, pages 53–96. Springer, 1995.

P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

J. Launchbury. A natural semantics for lazy evaluation. In *Proceedings of POPL 1993*, pages 144–154. ACM Press, 1993.

D. J. C. MacKay. Introduction to Monte Carlo methods. In M. I. Jordan, editor, *Learning and Inference in Graphical Models*. Kluwer, 1998. Paperback: *Learning in Graphical Models*, MIT Press.

J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988. Revised 2nd printing, 1998.

A. Pettorossi. A powerful strategy for deriving efficient programs by transformation. In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 273–281. ACM Press, 1984.

S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

A. Pfeffer. CTPPL: A continuous time probabilistic programming language. In C. Boutilier, editor, *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, pages 1943–1950, 2009.

D. Pollard. *A User's Guide to Measure Theoretic Probability*. Cambridge University Press, 2001.

J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM National Conference*, volume 2, pages 717–740. ACM Press, 1972.

C. Strachey and C. P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. Technical Monograph PRG-11, Programming Research Group, Oxford University Computing Laboratory, 1974.

L. Tierney. A note on Metropolis-Hastings kernels for general state spaces. *The Annals of Applied Probability*, 8(1):1–9, 1998.

P. L. Wadler. How to replace failure by a list of successes: A method for exception handling, backtracking, and pattern matching in lazy functional languages. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 113–128. Springer, 1985.

A. Usage Testing

Section 5.6 uses the function

$$usage :: Expr\ a \to Var\ b \to Usage$$

to test how many times an expression uses a variable. The return type *Usage* offers just three possibilities:

```
data Usage = Never | AtMostOnce | Unknown
  deriving (Eq, Ord)
```

For example, we want

```
usage (If (Var "b") (Var "x") (Var "x")) "x" = AtMostOnce
usage (Add          (Var "x") (Var "x")) "x" = Unknown
```

This contrast between If and Add indicates that we need two algebraic structures on the type *Usage*.

First, some *Usage* values *entail* others as propositions. For example, if v is never used, then v is used at most once. This entailment relation just happens to be a total order, so we define the operator $\leqslant$ to mean entailment, by **deriving** *Ord* above.

Second, when two subexpressions together produce a final outcome, the counts of how many times they use v add up, and our knowledge of the counts forms a commutative monoid. For example, suppose $e' = \mathsf{Add}\ e'_1\ e'_2$, and we know that e'_1 never uses v and e'_2 uses v at most once. Then we know that e' uses v at most once. If instead we only know that e'_1 and e'_2 each use v at most once, then all we know about e' is it uses v at most twice. That's not useful knowledge about e', so we might as well represent it as *Unknown*. We define the operator $\oplus$ to add up our knowledge in this way:

```
instance Monoid Usage where
  mempty       = Never
  Never ⊕ u    = u
  u ⊕ Never    = u
  _ ⊕ _        = Unknown
```

Armed with these two instances, we can define the *usage* function. It amounts to an abstract interpretation of expressions:

```
usage StdRandom _ = Never
usage (Lit _)   _ = Never
usage (Var v)   v' = if eq v v' then AtMostOnce
                                 else Never
usage (Let v e e') v' = usage e v' ⊕ if eq v v' then Never
                                                 else usage e' v'
usage (Neg e)   v = usage e v
usage (Exp e)   v = usage e v
usage (Log e)   v = usage e v
usage (Not e)   v = usage e v
usage (Add e1 e2) v = usage e1 v ⊕ usage e2 v
```

$$
\begin{aligned}
&\mathit{usage}\ (\mathsf{Less}\ e_1\ e_2)\ \ v &&= \mathit{usage}\ e_1\ v \oplus \mathit{usage}\ e_2\ v\\
&\mathit{usage}\ (\mathsf{If}\ e\ e_1\ e_2)\ \ v &&= \mathit{usage}\ e\ v \oplus \mathit{max}\ (\mathit{usage}\ e_1\ v)\\
& && \qquad\qquad\qquad\qquad\qquad (\mathit{usage}\ e_2\ v)
\end{aligned}
$$

The Var and Let cases above use the function *eq* to test the equality of two *Var*s whose types might differ:

$$
\begin{aligned}
&\mathit{eq} :: \mathit{Var}\ a \to \mathit{Var}\ b \to \mathit{Bool}\\
&\mathit{eq}\ (\mathsf{Real}\ v)\ \ (\mathsf{Real}\ w) = v \equiv w\\
&\mathit{eq}\ (\mathsf{Bool}\ v)\ (\mathsf{Bool}\ w) = v \equiv w\\
&\mathit{eq}\ _\qquad\quad\ _\qquad\quad\ = \mathit{False}
\end{aligned}
$$

To fit Haskell's type system better, we distinguish variables whose names are the same *String* if their types differ. For example, *extendEnv* in Section 2 treats Real "x" and Bool "x" as different variables that do not shadow each other's bindings. In other words, *Real* and *Bool* variables in our language reside in separate namespaces. Hence $\mathit{eq}\ (\mathsf{Real}\ \texttt{"x"})\ (\mathsf{Bool}\ \texttt{"x"}) = \mathit{False}$.

B. Compositional Density Calculator

$$
\begin{aligned}
&\mathit{extendSEnv} :: \mathit{Var}\ a \to \mathit{General}\ a \to \mathit{SEnv} \to \mathit{SEnv}\\
&\mathit{extendSEnv}\ v\ x\ \sigma = \sigma\ \{\\
&\quad \mathit{lookupSEnv} = \mathit{extendSEnv}'\ v\ x\ (\mathit{lookupSEnv}\ \sigma)\ \}\\
&\mathit{extendSEnv}' :: \mathit{Var}\ a \to \mathit{General}\ a \to (\forall b. \mathit{Var}\ b \to \mathit{General}\ b)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \to (\forall b. \mathit{Var}\ b \to \mathit{General}\ b)\\
&\mathit{extendSEnv}'\ (\mathsf{Real}\ v)\ x\ _\ (\mathsf{Real}\ v')\ \mid v \equiv v' = x\\
&\mathit{extendSEnv}'\ (\mathsf{Bool}\ v)\ x\ _\ (\mathsf{Bool}\ v')\ \mid v \equiv v' = x\\
&\mathit{extendSEnv}'\ _\qquad\quad _\ \sigma\ v' \qquad\qquad\quad = \sigma\ v'
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{extendList} :: \mathit{Int} \to a \to [a] \to [a]\\
&\mathit{extendList}\ i\ x\ \mathit{xs}\\
&\quad \mid i \equiv \mathit{length}\ \mathit{xs} = \mathit{xs} +\!\!+ [x]\\
&\quad \mid \mathit{otherwise}\quad\ = \mathit{error}\ (\texttt{"Expected length "} +\!\!+ \mathit{show}\ i +\!\!+\\
&\qquad\qquad\qquad\qquad\quad \texttt{", got "} +\!\!+ \mathit{show}\ (\mathit{length}\ \mathit{xs}))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{generalReal} :: (\mathit{DEnv} \to \mathit{Real}) \to \mathit{General}\ \mathit{Real}\\
&\mathit{generalReal}\ f = \mathit{General}\ \{\\
&\quad \mathit{gExpect}\ \ = \lambda\rho\ c.\, c\ (f\ \rho),\\
&\quad \mathit{gDensity} = [\,]\ \}\\
&\mathit{generalBool} :: (\mathit{DEnv} \to (\mathit{Bool} \to \mathit{Real}) \to \mathit{Real}) \to \mathit{General}\ \mathit{Bool}\\
&\mathit{generalBool}\ e = \mathit{General}\ \{\\
&\quad \mathit{gExpect}\ \ = e,\\
&\quad \mathit{gDensity} = [\lambda\rho\ t.\, e\ \rho\ (\lambda x.\, \mathbf{if}\ t \equiv x\ \mathbf{then}\ 1\ \mathbf{else}\ 0)]\ \}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{allocate} :: \mathit{Var}\ a \to \mathit{SEnv} \to (\mathit{SEnv}, a \to \mathit{DEnv} \to \mathit{DEnv})\\
&\mathit{allocate}\ v@(\mathsf{Real}\ _)\ \sigma =\\
&\quad \mathbf{let}\ i = \mathit{freshReal}\ \sigma\\
&\quad \mathbf{in}\ (\mathit{extendSEnv}\ v\ (\mathit{generalReal}\ (\lambda\rho.\, \mathit{lookupReal}\ \rho\ !!\ i))\\
&\qquad\qquad\qquad \sigma\ \{\mathit{freshReal} = i+1\},\\
&\qquad\ \lambda x\ \rho.\, \rho\ \{\mathit{lookupReal} = \mathit{extendList}\ i\ x\ (\mathit{lookupReal}\ \rho)\ \})\\
&\mathit{allocate}\ v@(\mathsf{Bool}\ _)\ \sigma =\\
&\quad \mathbf{let}\ i = \mathit{freshBool}\ \sigma\\
&\quad \mathbf{in}\ (\mathit{extendSEnv}\ v\ (\mathit{generalBool}\ (\lambda\rho\ c.\, c\ (\mathit{lookupBool}\ \rho\ !!\ i)))\\
&\qquad\qquad\qquad \sigma\ \{\mathit{freshBool} = i+1\},\\
&\qquad\ \lambda x\ \rho.\, \rho\ \{\mathit{lookupBool} = \mathit{extendList}\ i\ x\ (\mathit{lookupBool}\ \rho)\ \})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{general}\ \mathsf{StdRandom} = (\lambda_.\, \mathit{Never},\\
&\quad \lambda_.\, \mathit{General}\ \{\\
&\quad \mathit{gExpect}\ \ = \lambda_\ c.\, \textstyle\int_0^1 \lambda x.\, c\ x,\\
&\quad \mathit{gDensity} = [\lambda_\ t.\, \mathbf{if}\ 0 < t \wedge t < 1\ \mathbf{then}\ 1\ \mathbf{else}\ 0]\ \})\\
&\mathit{general}\ (\mathsf{Lit}\ x) = (\lambda_.\, \mathit{Never},\\
&\quad \lambda_.\, \mathit{generalReal}\ (\lambda_.\, \mathit{fromRational}\ x))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{general}\ (\mathsf{Var}\ v) = (\lambda v'.\, \mathbf{if}\ \mathit{eq}\ v\ v'\ \mathbf{then}\ \mathit{AtMostOnce}\ \mathbf{else}\ \mathit{Never},\\
&\quad \lambda\sigma.\, \mathit{lookupSEnv}\ \sigma\ v)\\
&\mathit{general}\ (\mathsf{Let}\ v\ e\ e') = (\lambda v'.\, u\ v' \oplus \mathbf{if}\ \mathit{eq}\ v\ v'\ \mathbf{then}\ \mathit{Never}\ \mathbf{else}\ u'\ v',\\
&\quad \lambda\sigma.\, \mathbf{let}\ (\sigma', \varepsilon) = \mathit{allocate}\ v\ \sigma\\
&\qquad\qquad\quad\ \sigma'' = \mathit{extendSEnv}\ v\ (g\ \sigma)\ \sigma\ \mathbf{in}\ \mathit{General}\ \{\\
&\quad \mathit{gExpect}\ \ = \lambda\rho\ c.\, \mathit{gExpect}\ (g\ \sigma)\ \rho\ (\lambda x.\\
&\qquad\qquad\qquad\qquad \mathit{gExpect}\ (g'\ \sigma')\ (\varepsilon\ x\ \rho)\ c),\\
&\quad \mathit{gDensity} = [\lambda\rho\ t.\, \mathit{gExpect}\ (g\ \sigma)\ \rho\ (\lambda x.\, \delta'\ (\varepsilon\ x\ \rho)\ t)\\
&\qquad\qquad\qquad \mid \delta' \leftarrow \mathit{gDensity}\ (g'\ \sigma')]\\
&\qquad\qquad\ +\!\!+ [\delta' \mid u'\ v \leqslant \mathit{AtMostOnce}\\
&\qquad\qquad\qquad\ , \delta' \leftarrow \mathit{gDensity}\ (g'\ \sigma'')]\ \})\\
&\quad \mathbf{where}\ (u\ \ , g\ \ ) = \mathit{general}\ e\\
&\qquad\qquad\ (u', g') = \mathit{general}\ e'\\
&\mathit{general}\ (\mathsf{Neg}\ e) = (u,\\
&\quad \lambda\sigma.\, \mathit{General}\ \{\\
&\quad \mathit{gExpect}\ \ = \lambda\rho\ c.\, \mathit{gExpect}\ (g\ \sigma)\ \rho\ (\lambda x.\, c\ (-x)),\\
&\quad \mathit{gDensity} = [\lambda\rho\ t.\, \delta\ \rho\ (-t) \mid \delta \leftarrow \mathit{gDensity}\ (g\ \sigma)]\ \})\\
&\quad \mathbf{where}\ (u, g) = \mathit{general}\ e\\
&\mathit{general}\ (\mathsf{Exp}\ e) = (u,\\
&\quad \lambda\sigma.\, \mathit{General}\ \{\\
&\quad \mathit{gExpect}\ \ = \lambda\rho\ c.\, \mathit{gExpect}\ (g\ \sigma)\ \rho\ (\lambda x.\, c\ (\mathit{exp}\ x)),\\
&\quad \mathit{gDensity} = [\lambda\rho\ t.\, \mathbf{if}\ 0 < t\ \mathbf{then}\ \delta\ \rho\ (\mathit{log}\ t)/t\ \mathbf{else}\ 0\\
&\qquad\qquad\qquad \mid \delta \leftarrow \mathit{gDensity}\ (g\ \sigma)]\ \})\\
&\quad \mathbf{where}\ (u, g) = \mathit{general}\ e\\
&\mathit{general}\ (\mathsf{Log}\ e) = (u,\\
&\quad \lambda\sigma.\, \mathit{General}\ \{\\
&\quad \mathit{gExpect}\ \ = \lambda\rho\ c.\, \mathit{gExpect}\ (g\ \sigma)\ \rho\ (\lambda x.\, c\ (\mathit{log}\ x)),\\
&\quad \mathit{gDensity} = [\lambda\rho\ t.\, \delta\ \rho\ (\mathit{exp}\ t) \times \mathit{exp}\ t \mid \delta \leftarrow \mathit{gDensity}\ (g\ \sigma)]\ \})\\
&\quad \mathbf{where}\ (u, g) = \mathit{general}\ e\\
&\mathit{general}\ (\mathsf{Not}\ e) = (u,\\
&\quad \lambda\sigma.\, \mathit{generalBool}\ (\lambda\rho\ c.\, \mathit{gExpect}\ (g\ \sigma)\ \rho\ (\lambda x.\, c\ (\mathit{not}\ x))))\\
&\quad \mathbf{where}\ (u, g) = \mathit{general}\ e\\
&\mathit{general}\ (\mathsf{Add}\ e_1\ e_2) = (\lambda v.\, u_1\ v \oplus u_2\ v,\\
&\quad \lambda\sigma.\, \mathit{General}\ \{\\
&\quad \mathit{gExpect}\ \ = \lambda\rho\ c.\, \mathit{gExpect}\ (g_1\ \sigma)\ \rho\ (\lambda x.\\
&\qquad\qquad\qquad\qquad \mathit{gExpect}\ (g_2\ \sigma)\ \rho\ (\lambda y.\, c\ (x+y))),\\
&\quad \mathit{gDensity} = [\lambda\rho\ t.\, \mathit{gExpect}\ (g_1\ \sigma)\ \rho\ (\lambda x.\, \delta_2\ \rho\ (t-x))\\
&\qquad\qquad\qquad \mid \delta_2 \leftarrow \mathit{gDensity}\ (g_2\ \sigma)]\\
&\qquad\qquad\ +\!\!+ [\lambda\rho\ t.\, \mathit{gExpect}\ (g_2\ \sigma)\ \rho\ (\lambda y.\, \delta_1\ \rho\ (t-y))\\
&\qquad\qquad\qquad \mid \delta_1 \leftarrow \mathit{gDensity}\ (g_1\ \sigma)]\ \})\\
&\quad \mathbf{where}\ (u_1, g_1) = \mathit{general}\ e_1\\
&\qquad\qquad\ (u_2, g_2) = \mathit{general}\ e_2\\
&\mathit{general}\ (\mathsf{Less}\ e_1\ e_2) = (\lambda v.\, u_1\ v \oplus u_2\ v,\\
&\quad \lambda\sigma.\, \mathit{generalBool}\ (\lambda\rho\ c.\, \mathit{gExpect}\ (g_1\ \sigma)\ \rho\ (\lambda x.\\
&\qquad\qquad\qquad\qquad \mathit{gExpect}\ (g_2\ \sigma)\ \rho\ (\lambda y.\, c\ (x < y)))))\\
&\quad \mathbf{where}\ (u_1, g_1) = \mathit{general}\ e_1\\
&\qquad\qquad\ (u_2, g_2) = \mathit{general}\ e_2\\
&\mathit{general}\ (\mathsf{If}\ e\ e_1\ e_2) = (\lambda v.\, u\ v \oplus \mathit{max}\ (u_1\ v)\ (u_2\ v),\\
&\quad \lambda\sigma.\, \mathit{General}\ \{\\
&\quad \mathit{gExpect}\ \ = \lambda\rho\ c.\, \mathit{gExpect}\ (g\ \sigma)\ \rho\ (\lambda b.\\
&\qquad\qquad\qquad\qquad \mathit{gExpect}\ ((\mathbf{if}\ b\ \mathbf{then}\ g_1\ \mathbf{else}\ g_2)\ \sigma)\ \rho\ c),\\
&\quad \mathit{gDensity} = [\lambda\rho\ t.\, \mathit{gExpect}\ (g\ \sigma)\ \rho\ (\lambda b.\\
&\qquad\qquad\qquad\qquad (\mathbf{if}\ b\ \mathbf{then}\ \delta_1\ \mathbf{else}\ \delta_2)\ \rho\ t)\\
&\qquad\qquad\qquad \mid \delta_1 \leftarrow \mathit{gDensity}\ (g_1\ \sigma), \delta_2 \leftarrow \mathit{gDensity}\ (g_2\ \sigma)]\ \})\\
&\quad \mathbf{where}\ (u\ \ , g\ \ ) = \mathit{general}\ e\\
&\qquad\qquad\ (u_1, g_1) = \mathit{general}\ e_1\\
&\qquad\qquad\ (u_2, g_2) = \mathit{general}\ e_2
\end{aligned}
$$