Submitted to ICFP 2015

Deriving a probability density calculator (functional pearl)

Wazim Mohammed Ismail Chung-chieh Shan
Indiana University
{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 output of our procedure 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 in this approach, we define a probability distribution, then answer questions about it such as "what is its expected value?" and "what does its histogram look like?". Over the course of 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 of modeling and inference constitute the beginning of a *combinator library* [6].

Unfortunately, models and inference procedures do not compose in tandem: as illustrated in Sections 3.1 and 6.2, often we rejoice that a big distribution we are interested in can be expressed naturally by composing little distributions, but then despair that many questions we want to pose about the big distribution cannot be answered using answers to corresponding questions about the little distributions. In other words, the natural compositional structure of models and 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 is like trying to build a compiler that generates an executable for a compound expression "$e_1;e_2;$" by combining the executables generated for the subexpressions "$e_1;$" and "$e_2;$".

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

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

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

Figure 1. The syntax and type system of our language of distributions

Still, there is 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. We define those interpreters by equational reasoning from a semantic specification. Our derivation demonstrates the power of combining λ-calculus with integral calculus.

2. A language of generative stories

To be concrete, we define a small language of distributions. To keep things simple, we include only two types in this language, *Real* and *Bool*. Figure 1 shows the syntax of our language. It defines a typing judgment $e : a$, which means as usual that the expression e has the type a.

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 is 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 add them to yield the final outcome. That final outcome 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*. 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 [8], it is 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.

Note that 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 add the chosen number to itself (in other words, double it) to yield the final outcome. In general, Let means to make a random choice once then use its outcome any number of times. Thus, we can understand this language as a call-by-value language whose side effect is random choice. In Let $v\ e\ e'$, the bound variable v takes scope over e' and not e.

In Haskell, we can define a data type *Expr* to represent the expressions of our language. Actually, using the GADT (generalized algebraic data type) extension, let us 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, Inv,
    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
```

We also define the GADT *Var* to represent variable names of each type.

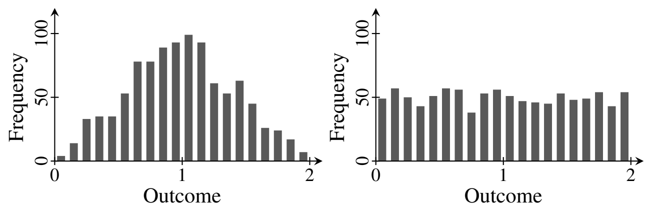
```
data Var a where
  Real :: String → Var Real
  Bool :: String → Var Bool
```

To fit Haskell’s type system better, we treat two variables whose types are different but whose names are the same *String* as different. For example, Real "x" and Bool "x" are different variables and do not shadow each other’s bindings. In other words, *Real* and *Bool* variables in our language reside in separate namespaces. We express this separation in our definition of the *jmEq* function, which checks if two *Var*s are equal, whether they have the same type.

```
jmEq :: Var a → Var b → Bool
jmEq (Real v) (Real w) = v ≡ w
jmEq (Bool v) (Bool w) = v ≡ w
jmEq _        _        = False
```

Hence *jmEq* (Real "x") (Bool "x") = *False*. For brevity, though, we elide applying Real and Bool to literal strings in examples.

As explained above, expressions in our language can be interpreted as generative stories. We can write an interpreter function to express this fact. This function *sample* 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 us 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:



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

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.

```
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 (Inv e)      ρ = liftM  recip  (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.

3. Composing expectation functionals

Although the *sample* interpreter is easy to write and intuitive to use, we should not think that the *IO* action it returns is equal to an expression's meaning. By "meaning" here, we mean what inference should preserve. The problem with treating *sample e* as the meaning of e is that we often want to optimize e to another expression e'. Usually, *sample* e' makes different and fewer random choices than *sample e*, so *sample* e' is different from *sample e*.

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

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, all the random choices in e_1 must be made before any of the random choices in e_2, even though the order does not matter. Thus, given that addition is commutative, we should assign Add e_1 e_2 and Add e_2 e_1 the same meaning.

Thus, the meaning equivalence relation produced by the *sample* semantics is too fine-grained. To make the equivalence coarser, let us consider the *expected values* of distributions. Given an expression of type *Real*, its expected value 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 do not 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—if we were to define a Haskell function

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

then it would not be straightforward the way *sample* is. For example, there is no way to define

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

in terms of *expect e*. People building a compiler for distributions, including the present authors, want compositionality in order to achieve separate compilation.

To make *expect* compositional, we add an argument to it to represent the context [5, 6] that an expression is plugged into before its expected value is observed. The type of *expect* is thus

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

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. (We assume c is measurable and non-negative, but do not worry if you are not familiar with such assumptions.)

Thanks to this generalization of *expect*, it is now compositional: we can define *expect* on an expression in terms of *expect* on its subexpressions. Here is the definition:

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

3.2 Integrals denoted by random choices

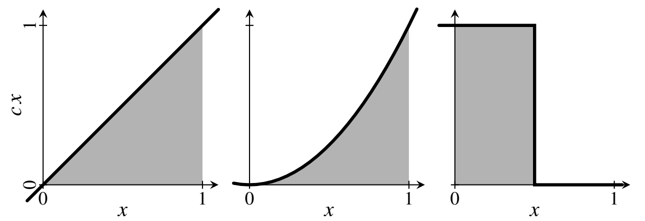
To understand this definition, let us start at the top.

The expected value of StdRandom is $1/2$, but that is 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 try to blend Haskell and mathematical notation by writing this integral as $\int_0^1 \lambda x.\, c\ x$, as if there is 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, symbolic integration, or mere printing of mathematical formulas (by overloading the *Num* class). (We also notate multiplication by $\times$, so $c\ x$ always means applying c to x, not multiplying c by x.)

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



$c\,x = x$ $\qquad$ $c\,x = x \times x$ $\qquad$ $c\,x = \textbf{if}\ x < 1/2\ \textbf{then}\ 1\ \textbf{else}\ 0$

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)

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

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

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

Figure 3 depicts these integrals.

The rest of the definition of *expect* is in continuation-passing style. The continuation c is the function whose expectation with respect to the current distribution we want. The Lit and Var cases are deterministic (that is, they do not make any random choices), so the expectation of c with respect to those distributions simply applies c to one value. The unary operators (Neg, Inv, 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

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

The order of nesting on the last line does not matter, as Tonelli's theorem assures us that it is 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 (which we assume of c). Thus we could define equivalently

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

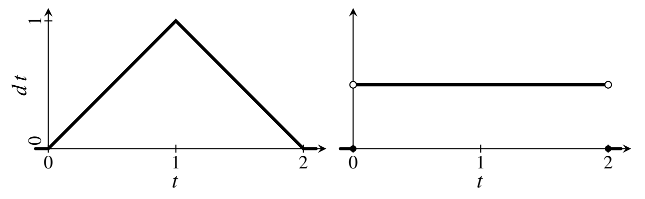
Besides compositionality, another benefit of generalizing *expect* is that it subsumes every question we can ask about a distribution. For example, if e is a closed expression of type *Real*, then the probability that the outcome of e is less than $1/2$ is

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

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

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

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



$$d\,t = \begin{cases} t & \text{if } 0 < t \le 1\\ 2-t & \text{if } 1 < t \le 2\\ 0 & \text{otherwise}\end{cases} \qquad d\,t = \begin{cases} 1/2 & \text{if } 0 < t < 2\\ 0 & \text{otherwise}\end{cases}$$

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

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 is injective. (In fact, it is bijective between measures and "increasing linear functionals with the Monotone Convergence property" [11, 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 optimize e and e' to each other.

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

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

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 $\lambda x. \textbf{if}\ x < 1/2\ \textbf{then}\ 1\ \textbf{else}\ 0$, 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 is part of 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. Without going into details, let us just say that density functions are very useful in probabilistic inference: they underpin many concepts and techniques, including *maximum-likelihood estimation*, *conditioning*, and *Monte Carlo sampling* [7, 13].

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 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 *reference measure*.

When the distribution is over reals, the reference measure is typically the Lebesgue measure over reals, and the definition amounts to the following.

Definition 1. A function $d :: \mathit{Real} \to \mathit{Real}$ is a *density* for a functional $m :: (\mathit{Real} \to \mathit{Real}) \to \mathit{Real}$ with respect to the Lebesgue measure if and only if

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

for all continuations $c :: \mathit{Real} \to \mathit{Real}$.

And when the distribution is over booleans, the reference measure is typically the counting measure over booleans, and the definition amounts to the following.

Definition 2. A function $d :: \mathit{Bool} \to \mathit{Real}$ is a *density* for a functional $m :: (\mathit{Bool} \to \mathit{Real}) \to \mathit{Real}$ with respect to the counting measure if and only if

$$m\ c = \mathit{sum}\ [d\ t \times c\ t \mid t \leftarrow [\mathit{True}, \mathit{False}]]$$

for all continuations $c :: \mathit{Bool} \to \mathit{Real}$.

In these definitions, the functional m might equal $\mathit{expect}\ e\ \rho$ for some expression e and environment ρ. Given e and ρ, because densities are useful, our goal is to find some function d that satisfies the specification above. To illustrate these definitions, let us check that the functions in Figure 4 are indeed densities of their respective distributions. First let us consider the function on the right of Figure 4, which is supposed to be a density for the functional

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

Here comes some equational reasoning 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 for m, 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 for the same m just as well.

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

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

It is again calculus time. 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 $\int_{-\infty}^{\infty} \cdot$):

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

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

Because density functions are useful, we want a program that automatically computes density functions from distribution expressions. Two such programs have been built before, but they "compute functions" in two different senses of the phrase. Pfeffer's [10, §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 [1, 2] density calculator deterministically produces an exact mathematical formula (which may contain integrals). 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 patently compositional density calculator. It produces a density function that can be treated both as an exact formula and as an approximation algorithm. We use $\int_\cdot^\cdot \cdot$ to integrate over reals, and perform usual operations on real numbers such as *negate* and *exp* (basically, the members of the Haskell type class *Floating* and its superclasses).

5. Calculating probability densities

To recap, our goal in the rest of this paper to write a program that, given e and ρ, finds a function d that satisfies Definitions 1 and 2 for $m = \mathit{expect}\ e\ \rho$.

Actually, such a function d does not always exist. For example, when $e = \mathsf{Lit}\ 3$, we want a function d such that

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

for all $c :: \mathit{Real} \to \mathit{Real}$. 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 $d :: \mathit{Real} \to \mathit{Real}$ is. So there is no such d.

Thus, not every distribution has a density. Moreover, not every density can be represented using the operations on *Real* available to us. So we have to relax our goal: let us write a program

$$\mathit{density} :: \mathit{Expr}\ a \to [\mathit{Env} \to a \to \mathit{Real}]$$

that maps each distribution expression e to a *list* of successes [14]. 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 the functional $\mathit{expect}\ e\ \rho$. In other words (expanding Definitions 1 and 2), for all δ, ρ, and c, the equation

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

(if $e :: \mathit{Expr}\ \mathit{Real}$) or the equation

$$\mathit{expect}\ e\ \rho\ c = \mathit{sum}\ [\delta\ \rho\ t \times c\ t \mid t \leftarrow [\mathit{True}, \mathit{False}]]$$

(if $e :: \mathit{Expr}\ \mathit{Bool}$) should hold.

The list returned by *density* might be empty, but we will do our best to keep it non-empty. For example, we regret that *density* (Lit 3) must be the empty list, but

$$\textit{density}\ (\mathsf{Let}\ \texttt{"x"}\ \mathsf{StdRandom}\ (\mathsf{Add}\ (\mathsf{Var}\ \texttt{"x"})\ (\mathsf{Var}\ \texttt{"x"})))$$

can be the non-empty list

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

as shown in Section 4.

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 $\lambda e.\,(\textit{density}\ e, \textit{expect}\ e)$ is compositional (but see Section 6.2).

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

5.1 Real base cases

An important base case is when $e = \mathsf{StdRandom}$: we define

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

This clause satisfies Definition 1 because

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

For the other base cases of type *Real*, we must regrettably fail, as just discussed.

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

5.2 Boolean cases

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

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

Matching the right-hand-side against Definition 2 shows that

$$\lambda t.\, m\ (\lambda x.\,\textbf{if}\ t \equiv x\ \textbf{then}\ 1\ \textbf{else}\ 0)$$

is a density for *m*. Accordingly, we define

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

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

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

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

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

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

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

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

Therefore, to match the goal, we define

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

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

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

and seek some δ' satisfying

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

Starting with the left-hand-side, we calculate

$$\begin{aligned} & \textit{expect}\ (\mathsf{Inv}\ e)\ \rho\ c \\ = \quad & \text{-- definition of } \textit{expect} \\ & \textit{expect}\ e\ \rho\ (\lambda x.\, c\ (1/x)) \\ = \quad & \text{-- induction hypothesis, substituting } \lambda x.\, c\ (1/x) \text{ for } c \\ & \textstyle\int_{-\infty}^{\infty} \lambda x.\,\delta\ \rho\ x \times c\ (1/x) \\ = \quad & \text{-- changing the integration variable from } x \text{ to } t = 1/x \\ & \textstyle\int_{-\infty}^{\infty} \lambda t.\,(\delta\ \rho\ (1/t)/t/t) \times c\ t \end{aligned}$$

(At the last step, the factor $1/t/t$ is the absolute value of the derivative of $x = 1/t$ with respect to t.) Therefore, to match the goal, we define

$$\textit{density}\ (\mathsf{Inv}\ e) = [\lambda \rho\ t.\,\delta\ \rho\ (1/t)/t/t \mid \delta \leftarrow \textit{density}\ e]$$

The case *density* (Exp *e*) illustrates another slight complication: because the result of exponentiation is never negative, our derivation prompts us to take the domain of integration into account. We seek δ' such that

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

so we calculate

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

(At the second-to-last step, the factor $1/t$ is the absolute value of the derivative of $x = \textit{log}\ t$ with respect to t.) Therefore, to match the goal, we define

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

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

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

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 do not need a density for the subexpression e.) We seek some δ' such that the equation

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

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 $\lambda c.\,\mathit{sum}\ (\mathit{map}\ c\ [\mathit{True}, \mathit{False}])$ (if e_1 and e_2 have type *Expr Bool*). Starting with the left-hand-side, we calculate

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

Therefore, to match the goal, we define

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

5.5 Binary operators

Binary operators bring a new twist to our derivation, namely that our density calculator can be *nondeterministic*: it can try multiple strategies for finding a density, and if multiple strategies succeed, the results are equivalent.

Take Add $e_1\ e_2$ for example. The distribution denoted by 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

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

holds for all ρ and c.

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

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

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

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

Therefore, we can define

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

By analogous reasoning, we can also define

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

Although these two lists can overlap (for example when e_1 and e_2 are both StdRandom), they do not subsume each other. For example, because Lit 3 has no density, only the first definition handles Add (Lit 3) StdRandom and only the second definition handles Add StdRandom (Lit 3). In the end, we define

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

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, just like in the good old slide-rule days.)

5.6 Variable binding and sharing

As with Add, an expression Let $v\ e\ e'$ may have a density even if one of its subexpressions e and e' does not. We call v the bound variable, e the *right-hand-side*, and e' the *body* of the Let. There are two basic strategies for handling Let.

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

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

Therefore, we can define

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

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)
```

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

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

by turning it into the equivalent expression

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

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

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

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

In general, Tonelli's theorem tells us that delayed evaluation preserves the expectation semantics of the expression $\mathsf{Let}\ v\ e\ e'$ 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 $expect\ e'\ (extendEnv\ v\ x\ \rho)\ c = expect\ e'\ \rho\ c$, so

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

and a simple induction on e shows that $expect\ e\ \rho\ (\lambda x.\, 1)$ is always equal to 1 in our language.

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

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

5.6.1 Usage testing

The condition $usage\ e'\ v \leqslant AtMostOnce$ above tests conservatively whether the expression e' uses the variable v at most once. (This test serves the purpose of Bhat et al.'s [1] *active variables* and independence test.) The rest of this section describes how we define this test. You can skip to the next section, but then you would miss a nice example of an order and a monoid used to define an abstract interpretation.

The type of *usage* is

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

The return type *Usage* represents our knowledge about how many times the expression e' uses the variable v.

$$\begin{aligned} &\textbf{data}\ Usage = Never \mid AtMostOnce \mid Unknown \\ &\quad \textbf{deriving}\ (Eq, Ord) \end{aligned}$$

The type *Usage* has two useful algebraic structures. 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 is 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:

$$\begin{aligned} &\textbf{instance}\ Monoid\ Usage\ \textbf{where} \\ &\quad mempty \qquad\quad = Never \\ &\quad Never \oplus u \qquad = u \\ &\quad u \qquad \oplus Never = u \\ &\quad \_ \qquad\ \oplus \_ \qquad = Unknown \end{aligned}$$

Armed with these two instances, we can define the *usage* function:

$$\begin{aligned} &usage\ \mathsf{StdRandom}\ \_ = Never \\ &usage\ (\mathsf{Lit}\ \_) \qquad\ \_ = Never \\ &usage\ (\mathsf{Var}\ v) \qquad v' = \textbf{if}\ jmEq\ v\ v'\ \textbf{then}\ AtMostOnce \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ Never \\ &usage\ (\mathsf{Let}\ v\ e\ e')\ \ v' = usage\ e\ v' \oplus \textbf{if}\ jmEq\ v\ v'\ \textbf{then}\ Never \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ usage\ e'\ v' \\ &usage\ (\mathsf{Neg}\ e) \qquad\ v\ = usage\ e\ v \\ &usage\ (\mathsf{Inv}\ e) \qquad\ \ v\ = usage\ e\ v \\ &usage\ (\mathsf{Exp}\ e) \qquad\ v\ = usage\ e\ v \\ &usage\ (\mathsf{Log}\ e) \qquad\ v\ = usage\ e\ v \\ &usage\ (\mathsf{Not}\ e) \qquad\ v\ = usage\ e\ v \\ &usage\ (\mathsf{Add}\ e_1\ e_2)\ \ v\ = usage\ e_1\ v \oplus usage\ e_2\ v \\ &usage\ (\mathsf{Less}\ e_1\ e_2)\ v\ = usage\ e_1\ v \oplus usage\ e_2\ v \\ &usage\ (\mathsf{If}\ e\ e_1\ e_2)\ \ v\ = usage\ e\ v \oplus max\ (usage\ e_1\ v) \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (usage\ e_2\ v) \end{aligned}$$

5.6.2 Monad laws

Every monad is supposed to obey the three laws [15]

$$\begin{aligned} &return\ a \gg\!= k \qquad\quad = k\ a \qquad\qquad\qquad \text{-- Left unit} \\ &m \gg\!= return \qquad\ \ = m \qquad\qquad\qquad\ \ \text{-- Right unit} \\ &m \gg\!= \lambda a.\, (k\ a \gg\!= h) = (m \gg\!= k) \gg\!= h \quad \text{-- Associativity} \end{aligned}$$

if only up to observation [5]. Our language has random choice as an implicit side effect, so there are no explicit constructs *return* and $\gg\!=$, but as in a typical call-by-value language, the meaning of our Let is the $\gg\!=$ operation of the underlying probability monad [12]. Accordingly, we want the following three laws to hold, if only up to observation:

$$\begin{aligned} &\mathsf{Let}\ v\ (\mathsf{Var}\ v)\ e \qquad\ = e \\ &\mathsf{Let}\ v\ e\ (\mathsf{Var}\ v) \qquad\ = e \\ &\mathsf{Let}\ v_2\ (\mathsf{Let}\ v_1\ e_1\ e_2)\ e = \mathsf{Let}\ v_1\ e_1\ (\mathsf{Let}\ v_2\ e_2\ e) \\ &\qquad\qquad\qquad\qquad\qquad\quad \text{-- if } usage\ e\ v_1 \equiv Never \end{aligned}$$

It is easy to check that these equations hold under the observation functions *sample* and *expect*. For example, it is easy to check that

$$\begin{aligned} &\quad expect\ (\mathsf{Let}\ v\ e\ (\mathsf{Var}\ v)) \\ = &\quad \text{-- definition of } expect \\ &\quad \lambda \rho\ c.\, expect\ e\ \rho\ (\lambda x.\, expect\ (\mathsf{Var}\ v)\ (extendEnv\ v\ x\ \rho)\ c) \\ = &\quad \text{-- definition of } expect,\ lookupEnv, \text{ and } extendEnv \\ &\quad \lambda \rho\ c.\, expect\ e\ \rho\ (\lambda x.\, c\ x) \\ = &\quad \text{-- } \eta\text{-reductions} \\ &\quad expect\ e \end{aligned}$$

A more interesting exercise is to check that the same equations hold under the observation functions *density*, as long as we treat the list of successes returned by *density* as a set. We leave this to the reader.

6. Properties of our density calculator

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

6.1 Incompleteness

As shown in Section 4, the distribution

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

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

$$\mathit{density}\ (\mathsf{Let}\ \texttt{"x"}\ \mathsf{StdRandom}\ (\mathsf{Add}\ (\mathsf{Var}\ \texttt{"x"})\ (\mathsf{Var}\ \texttt{"x"})))$$

were to return the non-empty list

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

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.

6.2 Compositionality

As promised above Section 5.1, our definition of *density e* not only uses the *density* of the subexpressions of *e*, but also uses *expect*. After all, we have seen that *density* itself is not compositional. 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 the expected value of a distribution 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

```
data GeneralDensity a = GD {
  gdExpect  :: Env → (a → Real) → Real,
  gdUsage   :: ∀b.Var b → Usage,
  gdDensity :: [Env → a → Real]}
generalDensity :: Expr a → GeneralDensity a
generalDensity e = GD {
  gdExpect  = expect e,
  gdUsage   = usage e,
  gdDensity = density e}
```

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

```
generalDensity (Add e₁ e₂) = GD {
  gdExpect  = λρ c.gdExpect gd₁ ρ (λx.
                   gdExpect gd₂ ρ (λy.c (x+y))),
  gdUsage   = λv.gdUsage gd₁ v ⊕ gdUsage gd₂ v,
  gdDensity = [λρ t.gdExpect gd₁ ρ (λx.δ₂ ρ (t−x))
              | δ₂ ← gdDensity gd₂]
           ++ [λρ t.gdExpect gd₂ ρ (λy.δ₁ ρ (t−y))
              | δ₁ ← gdDensity gd₁]}
  where gd₁ = generalDensity e₁
        gd₂ = generalDensity e₂
```

collecting the definition of *expect* (Add e_1 e_2) in Section 3.1, the definition of *usage* (Add e_1 e_2) in Section 5.6.1, and the definition of *density* (Add e_1 e_2) in Section 5.5. This is the *tupling transformation* [3, 9] applied to the pattern of *dependent interpretations* discussed by Gibbons and Wu [4, §4.2].

The use of *density* $(e'\{v \mapsto e\})$ to define *density* (Let v e e') complicates our quest for compositionality, because the recursive argument $e'\{v \mapsto e\}$ is not necessarily a subexpression of 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.

```
data SEnv = SEnv {
  freshReal   :: Int,
  freshBool   :: Int,
  lookupSEnv :: ∀a.Var a → General a}
data General a = General {
  gExpect    :: DEnv → (a → Real) → Real,
  gDensity   :: [DEnv → a → Real]}
data DEnv = DEnv {
  lookupReal :: [Real],
  lookupBool :: [Bool]}
```

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.

```
emptySEnv :: SEnv
emptySEnv = SEnv {freshReal   = 0,
                  freshBool   = 0,
                  lookupSEnv = λv.error "Unbound"}
emptyDEnv :: DEnv
emptyDEnv = DEnv {lookupReal = [],
                  lookupBool = []}
```

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 the appendix.

```
general :: Expr a → (∀b.Var b → Usage,
                     SEnv → General a)
```

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

```
runDensity :: Expr a → [a → Real]
runDensity e = [δ emptyDEnv
               | δ ← gDensity (snd (general e) emptySEnv)]
```

7. Approximating probability densities

The density calculator derived in Section 5 produces output rife with integrals. The definition of *density* itself does not contain integrals, but *expect* StdRandom contains an integral, and *density* is defined in terms of *expect* in the boolean, If, Add, and Let cases. For example, here is one success of our density calculator:

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

One way to use *density* is to feed its output to a computer algebra system for simplification. If we are 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.

Moreover, even if some integrals cannot be simplified away, we can execute the function produced by *density* 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. For example, we can interpret the successful result above, as is, as the following randomized (and embarrassingly parallel) algorithm:

Algorithm 1. Given t, choose a random real number x uniformly between 0 and 1, then compute $\textbf{if}\ 0 < t - x < 1\ \textbf{then}\ 1\ \textbf{else}\ 0$.

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

A more substantial example is the distribution

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

This input exercises the nondeterminism in the Add case of *density*:

$$\begin{aligned}
&\textit{density}\ (\mathsf{Add\ StdRandom\ (Exp\ (Neg\ StdRandom))})\\
&= [\lambda\rho\ t.\,\textit{expect}\ \mathsf{StdRandom}\ \rho\ (\lambda x.\,\delta_2\ \rho\ (t-x))\\
&\qquad \mid \delta_2 \leftarrow \textit{density}\ (\mathsf{Exp\ (Neg\ StdRandom)})]\\
&\;\; +\!\!+ [\lambda\rho\ t.\,\textit{expect}\ (\mathsf{Exp\ (Neg\ StdRandom)})\ \rho\ (\lambda y.\,\delta_1\ \rho\ (t-y))\\
&\qquad \mid \delta_1 \leftarrow \textit{density}\ \mathsf{StdRandom}]\\
&= [\lambda\rho\ t.\textstyle\int_0^1 \lambda x.\,\textbf{if}\ 0 < t-x\\
&\qquad\qquad\qquad \textbf{then}\ (\textbf{if}\ 0 < -\textit{log}\ (t-x) < 1\ \textbf{then}\ 1\ \textbf{else}\ 0)\\
&\qquad\qquad\qquad\qquad /(t-x)\\
&\qquad\qquad\qquad \textbf{else}\ 0,\\
&\quad\ \lambda\rho\ t.\textstyle\int_0^1 \lambda z.\,\textbf{if}\ 0 < t - \textit{exp}\ (-z) < 1\ \textbf{then}\ 1\ \textbf{else}\ 0]
\end{aligned}$$

Suppose we can simplify these two results no further. Nevertheless, they can be interpreted as two randomized algorithms:

Algorithm 2. Given t, choose x between 0 and 1, then compute

$$\begin{aligned}
&\textbf{if}\ 0 < t-x\\
&\textbf{then}\ (\textbf{if}\ 0 < -\textit{log}\ (t-x) < 1\ \textbf{then}\ 1\ \textbf{else}\ 0)\\
&\qquad /(t-x)\\
&\textbf{else}\ 0
\end{aligned}$$

In short, sample x from the first summand $\mathsf{StdRandom}$, then compute the density of the second summand $\mathsf{Exp\ (Neg\ StdRandom)}$ at $t-x$.

Algorithm 3. Given t, choose z between 0 and 1, then compute

$$\textbf{if}\ 0 < t - \textit{exp}\ (-z) < 1\ \textbf{then}\ 1\ \textbf{else}\ 0$$

In other words, sample $y = \textit{exp}\ (-z)$ from the second summand $\mathsf{Exp\ (Neg\ StdRandom)}$, then compute the density of the first summand $\mathsf{StdRandom}$ at $t-y$.

Both algorithms are correct, in the sense that the expected output from each algorithm is an actual density at t. Therefore, we can estimate a density by running the algorithms many times and averaging the results when time is about to run out. In general, it is correct in this sense to interpret each call from *density* to *expect e* as sampling randomly from *e*. In particular, it is correct to interpret $\int_0^1 \lambda x$ as choosing x uniformly between 0 and 1, because the result of the density formula is always *affine* in the result of the integral. That is, integrals appear only in positions such as

$$\frac{\int\cdots}{\cdots} \quad\text{and}\quad \frac{1}{2} + \int\cdots, \qquad\text{but not}\qquad \frac{\cdots}{\int\cdots} \quad\text{or}\quad \left(\int\cdots\right)^2.$$

In fact, we can randomly choose between the two algorithms on each iteration, and this probabilistic mixture of the two algorithms is also correct. That is what Pfeffer's [10, §5.2] approximate algorithm for density estimation does: each time it encounters an expression of the form $\mathsf{Add}\ e_1\ e_2$, it randomly chooses whether to sample from e_1 then attempt to compute the density of e_2, or to sample from e_2 then attempt to compute the density of e_1. If the attempt fails, then the algorithm just produces no density estimate on that particular iteration (which is different from estimating 0). Although possibly less accurate, this randomization brings several potential advantages:

1. It does not spend time trying symbolic integration.
2. It does not need to analyze the entire input expression before starting to generate density estimates. This is especially suitable for a pipelined setting, where the input expression may be generated or unrolled on the fly.
3. It may succeed on some input expressions containing unreached subexpressions that stymie the exact algorithm.

8. Making density approximation more accurate

We have seen that the formula produced by our density calculator not only denotes an exact mathematical function but also can be interpreted as an approximation algorithm. Sometimes we can simplify the integrals in the formula away and get a closed form that runs in constant time. But even if we cannot eliminate all integrals, simplifying the density formula can make the corresponding approximation algorithm run faster and produce results that vary less from run to run, and so yield a more accurate density estimate given the same amount of time.

For example, our density calculator succeeds on the expression

$$\begin{aligned}
&\mathsf{If\ (Less\ StdRandom\ (Lit\ (1/2)))}\\
&\quad \mathsf{(Add\ StdRandom\ StdRandom)}\\
&\quad \mathsf{(Add\ StdRandom\ (Exp\ (Neg\ StdRandom)))}
\end{aligned}$$

in two ways. In other words, it produces two approximate algorithms for this expression's density. Both algorithms are easy to describe in terms of the building blocks in Section 7.

A. Flip a fair coin to choose between Algorithms 1 and 2.

B. Flip a fair coin to choose between Algorithms 1 and 3.

The experimental probabilistic programming system Hakaru simplifies these algorithms to

A simplified. Flip a fair coin to choose between

$$\textbf{if}\ 0 < t \leqslant 1\ \textbf{then}\ t\ \textbf{else if}\ 1 < t \leqslant 2\ \textbf{then}\ 2 - t\ \textbf{else}\ 0$$

and Algorithm 2.

B simplified. Flip a fair coin to choose between

$$\textbf{if}\ 0 < t \leqslant 1\ \textbf{then}\ t\ \textbf{else if}\ 1 < t \leqslant 2\ \textbf{then}\ 2 - t\ \textbf{else}\ 0$$

and Algorithm 3.

Hence the variance in Algorithms 2 and 3 remains, but the variance in Algorithm 1 is gone.

Using Hakaru, we ran these 4 algorithms (A and B, unsimplified and simplified) on a typical compute node. For each value of t in $[0, 0.02\,..\,2]$, we ran each algorithm 100 times, each time using 5 milliseconds of unparallelized CPU time and achieving several hundred iterations.

Figure 5 plots the standard deviation of the density estimates produced by each algorithm. Given that all 4 algorithms have the same correct expected value, lower in the plot is better because it means the algorithm's estimate is less variable and more accurate.

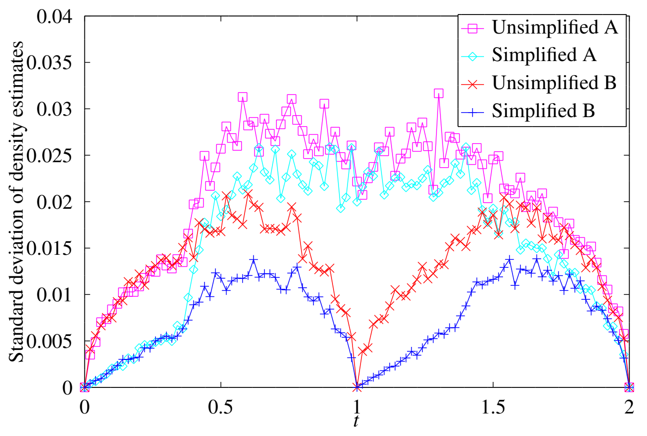


Figure 5. Standard deviation of the 100 density estimates produced by each of 4 different algorithms for each input value t

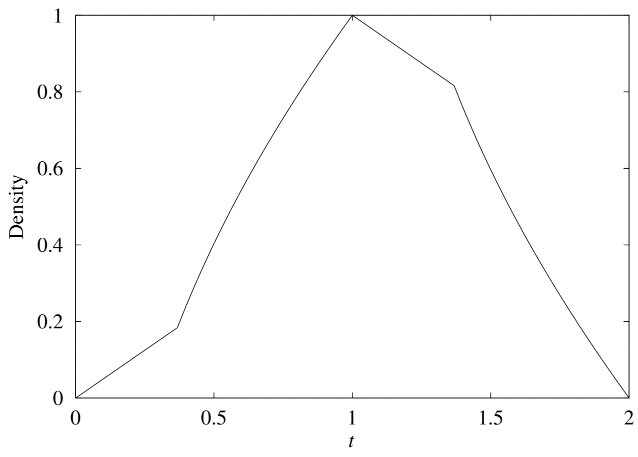


Figure 6. True density of the example distribution in Section 8

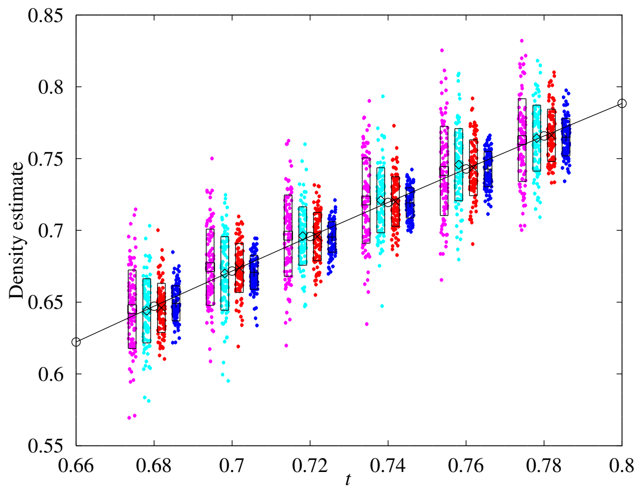


Figure 7. Scatter plot of density estimates for a typical region of input values. Each group of 4 vertical clusters represents the 100 density estimates produced by each algorithm for *one* input value t. Each cluster in the group of 4 represents a different algorithm (ordered and colored as in Figure 5). In each cluster, the box shows the mean and standard deviation, and the horizontal location of a point is not meaningful. The diagonal curve is the true density.

The plot shows that Algorithm B is better than Algorithm A, and more importantly, simplification improves both algorithms.

For reference, Figure 6 shows the true density of our example distribution. Figure 7 zooms into a typical region of the distribution ($t \in [0.68, 0.7 \,..\, 0.78]$) and plots each density estimate as a point. The more tightly the points are clustered together, the better the algorithm. We see that, in this region of the distribution, the algorithms from best to worst are Algorithm B simplified, Algorithm B unsimplified, Algorithm A simplified, Algorithm A unsimplified.

In sum, simplifying the output of our density calculator can make it more accurate even if some integrals remain. This hybrid between exact and approximate density computation is possible thanks to the mathematical semantics of both kinds of computation.

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 may be powerful ways to turn expressive probabilistic models into effective inference procedures. We are investigating this hypothesis in ongoing work.

Acknowledgments

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

[1] 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.

[2] 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: 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, number 7795 in Lecture Notes in Computer Science, pages 508–522. Springer, 2013.

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

[4] 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.

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

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

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

[12] N. Ramsey and A. Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *Proceedings of POPL 2002*, pages 154–165. ACM Press, 2002.

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

[14] 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.

[15] P. L. Wadler. The essence of functional programming. In *Proceedings of POPL 1992*, pages 1–14. ACM Press, 1992.

A. Compositional density calculator

```
extendSEnv :: Var a → General a → SEnv → SEnv
extendSEnv v x σ = σ {
  lookupSEnv = extendSEnv′ v x (lookupSEnv σ) }
extendSEnv′ :: Var a → General a → (∀b.Var b → General b)
                                 → (∀b.Var b → General b)
extendSEnv′ (Real v) x _ (Real v′) | v ≡ v′ = x
extendSEnv′ (Bool v) x _ (Bool v′) | v ≡ v′ = x
extendSEnv′ _        _ σ v′                 = σ v′

extendList :: Int → a → [a] → [a]
extendList i x xs
  | i ≡ length xs = xs ++ [x]
  | otherwise     = error ("Expected length " ++ show i ++
                           ", got " ++ show (length xs))

generalReal :: (DEnv → Real) → General Real
generalReal f = General {
  gExpect  = λρ c.c (f ρ),
  gDensity = [ ] }
generalBool :: (DEnv → (Bool → Real) → Real) → General Bool
generalBool e = General {
  gExpect  = e,
  gDensity = [λρ t.e ρ (λx.if t ≡ x then 1 else 0)] }

allocate :: Var a → SEnv → (SEnv, a → DEnv → DEnv)
allocate v@(Real _) σ =
  let i = freshReal σ
  in (extendSEnv v (generalReal (λρ.lookupReal ρ !! i))
                   σ {freshReal = i + 1},
      λx ρ.ρ {lookupReal = extendList i x (lookupReal ρ)})
allocate v@(Bool _) σ =
  let i = freshBool σ
  in (extendSEnv v (generalBool (λρ c.c (lookupBool ρ !! i)))
                   σ {freshBool = i + 1},
      λx ρ.ρ {lookupBool = extendList i x (lookupBool ρ)})

general StdRandom = (λ_.Never,
  λ_.General {
  gExpect  = λ_ c.∫_0^1 λx.c x,
  gDensity = [λ_ t.if 0 < t ∧ t < 1 then 1 else 0]})
general (Lit x) = (λ_.Never,
  λ_.generalReal (λ_.fromRational x))
general (Var v) = (λv′.if jmEq v v′ then AtMostOnce else Never,
  λσ.lookupSEnv σ v)
general (Let v e e′) = (λv′.u v′ ⊕ if jmEq v v′ then Never else u′ v′,
  λσ.let (σ′, ε) = allocate v σ
         σ″ = extendSEnv v (g σ) σ in General {
  gExpect  = λρ c.gExpect (g σ) ρ (λx.
                    gExpect (g′ σ′) (ε x ρ) c),
  gDensity = [λρ t.gExpect (g σ) ρ (λx.δ′ (ε x ρ) t)
               | δ′ ← gDensity (g′ σ′)]
             ++ [δ′ | u′ v ⩽ AtMostOnce
                    , δ′ ← gDensity (g′ σ″)]})
  where (u , g ) = general e
        (u′, g′) = general e′
general (Neg e) = (u,
  λσ.General {
  gExpect  = λρ c.gExpect (g σ) ρ (λx.c (−x)),
  gDensity = [λρ t.δ ρ (−t) | δ ← gDensity (g σ)]})
  where (u, g) = general e
general (Inv e) = (u,
  λσ.General {
  gExpect  = λρ c.gExpect (g σ) ρ (λx.c (1/x)),
  gDensity = [λρ t.δ ρ (1/t)/t/t | δ ← gDensity (g σ)]})
  where (u, g) = general e
general (Exp e) = (u,
  λσ.General {
  gExpect  = λρ c.gExpect (g σ) ρ (λx.c (exp x)),
  gDensity = [λρ t.if 0 < t then δ ρ (log t)/t else 0
               | δ ← gDensity (g σ)]})
  where (u, g) = general e
general (Log e) = (u,
  λσ.General {
  gExpect  = λρ c.gExpect (g σ) ρ (λx.c (log x)),
  gDensity = [λρ t.δ ρ (exp t) × exp t | δ ← gDensity (g σ)]})
  where (u, g) = general e
general (Not e) = (u,
  λσ.generalBool (λρ c.gExpect (g σ) ρ (λx.c (not x))))
  where (u, g) = general e
general (Add e1 e2) = (λv.u1 v ⊕ u2 v,
  λσ.General {
  gExpect  = λρ c.gExpect (g1 σ) ρ (λx.
                   gExpect (g2 σ) ρ (λy.c (x + y))),
  gDensity = [λρ t.gExpect (g1 σ) ρ (λx.δ2 ρ (t − x))
               | δ2 ← gDensity (g2 σ)]
             ++ [λρ t.gExpect (g2 σ) ρ (λy.δ1 ρ (t − y))
               | δ1 ← gDensity (g1 σ)]})
  where (u1, g1) = general e1
        (u2, g2) = general e2
general (Less e1 e2) = (λv.u1 v ⊕ u2 v,
  λσ.generalBool (λρ c.gExpect (g1 σ) ρ (λx.
                     gExpect (g2 σ) ρ (λy.c (x < y)))))
  where (u1, g1) = general e1
        (u2, g2) = general e2
general (If e e1 e2) = (λv.u v ⊕ max (u1 v) (u2 v),
  λσ.General {
  gExpect  = λρ c.gExpect (g σ) ρ (λb.
                   gExpect ((if b then g1 else g2) σ) ρ c),
  gDensity = [λρ t.gExpect (g σ) ρ (λb.
                   (if b then δ1 else δ2) ρ t)
               | δ1 ← gDensity (g1 σ), δ2 ← gDensity (g2 σ)]})
  where (u , g ) = general e
        (u1, g1) = general e1
        (u2, g2) = general e2
```