Expanding the Hakaru language to investigate probabilistic program transformation

Justin Staples, Mahmoud Khattab, Nevin Mahilal, Arian Sohrabi

April 26, 2018

Abstract. We present our work on Hakaru, a probabilistic programming language that has been expanded with the addition of new features and built in functions. Development on this project can be divided into two major endeavours. First, we present a strategy for developing implementations of new statistical distributions using the primitives that already existed in Hakaru. Known relationships between distributions are then used to test the validity of the Hakaru language. A small subset of the newly added disbributions have been tested in this way with mostly failing results.

1 Introduction

Often, we wish to create a model for some kind of real world phenomenon so that we can extract useful information and learn more about it. For example, in a machine learning application, the goal is often to predict or infer information about the model based on the past outcomes of some kind of experiment. An experiment could result in many different outcomes, each one with a different likelihood (probability).

The mathematical functions that we use to describe these probabilties are called probability distributions and the quantity that denotes the outcome of the experiment is called a random variable. A random variable can take on continuous or discrete values depending on the application. Continuous random variables are formally defined by a probability density function (PDF), which maps the outcomes of the random variable to real numbers that represent a probability. Similarly, a discrete random variable is described by a probability mass function (PMF). As an example, consider one of the most ubiquitous and naturally occuring distributions, the normal distribution. We can say that a random variable, X, is normally distributed with the following notation.

$$\boxed{X \sim Normal(\mu, \sigma^2)}$$

Here, we can see that the normal distribution is parametrized by μ, the population mean, and σ, the standard deviation. A plot the PDF is shown below. Remember that the PDF describes the liklihood that a value sampled from this distribution will equal x.

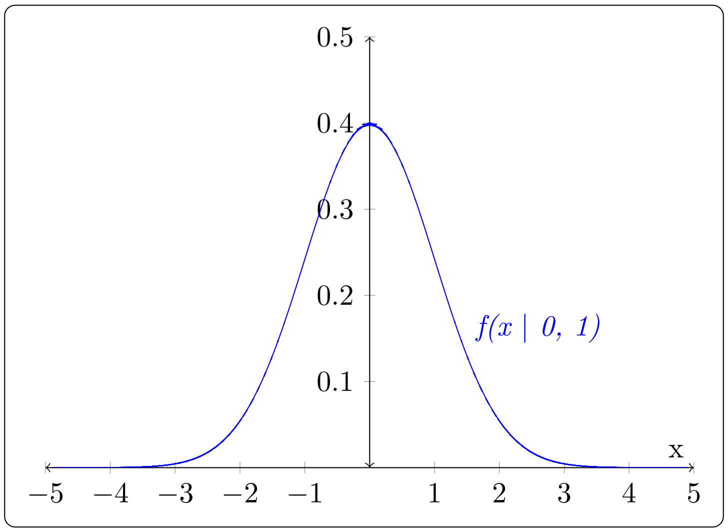

Figure 1: PDF of the normal distribution, with $\mu = 0$ and $\sigma = 1$

Hakaru [1] is an experimental probabilistic programming language (PPL) which allows users to create probabilistic models (implementations of statistical distributions). Running a Hakaru program generates a stream of random numbers (known as samples) distributed according to the distribution defined in the program code. The language is quite small given its specialized domain. However, Hakaru programs can be compiled to C and Haskell, making for an easy way to export probabilistic models defined in Hakaru into imperative and functional programming domains. These can then be used within larger applications (e.g. simulating scenarios with inherent uncertainty, generating training sets for machine learning algorithms).

The purpose of Hakaru is to simplify the process of implementing efficient probabilistic models. The small size of the language solves the problem of simplifying model implementation. In order to simplify the process of making these implementations efficient, Hakaru includes some inference algorithms that transform Hakaru programs into other forms. Most pertinent to our project is `hk-maple` (formerly known as `simplify`). In short, `hk-maple` takes a Hakaru program file as an argument and uses Maple to perform algebraic transformations. If the simplify mode of `hk-maple` is used (the default mode), it returns an equivalent Hakaru program with greater sampling efficiency. In applications which may require billions of samplings, such as machine learning, this has the potential to save a significant amount of processing time.

2 Motivation

As a probabilistic programming language, Hakaru is well suited for machine learning applications and problems that have inherent uncertainty. The workflow of many of these problems have elements in common. Hakaru was originally designed with ease of use in mind, meaning that it aims to streamline the process of designing solutions for these types of problems. The system description of Hakaru presents a new, modular approach to designing these solutions such that the solutions can be reused and composed with others. Studying the nature of the Hakaru language can reveal useful information about how these probabilistic models can be used to solve problems.

The purpose of our project is two-fold. Our first objective is to increase the accessibility to Hakaru for future users. Our main effort in this regard has been the development of a standard library in which we have implemented over 60 statistical distributions which cover a wide range of potential applications. Another effort to increase accessibility has been the development of a syntax highlighting package for Sublime Text; the only one currently implemented for a GUI text editor. This package has been invaluable in increasing code readability/writability, and optimizing our workflow. Lastly, a few primitive mathematical functions have been newly incorporated in to the language (e.g. `choose`, `log`, `sin`, etc.).

Our second objective is to test the validity of the Hakaru language. Our focus is on writing test cases based on well-known relationships between the various distributions implemented in the Standard Library. In general, our test cases transform two related distributions so that the resulting distributions are (hypothetically) equivalent to each other. The number of possible test cases we could write in this vein is beyond the scope of this project. To this end, we have focused our testing on a few distributions, with the intention of laying down the groundwork for future testing in this area.

3 Related Works

One of the major focuses of our project has been on testing relationships between distributions. Our team has been adding our testing files to a large testing suite written in Haskell (namely `RoundTrip.hs`, where all the tests are organized). This file included hundreds of tests, many of which were written by other Hakaru developers! So, this gives evidence that these types of program transformations have been investigated before. Of course, this work is very related and relelvant to our own. As a result, we have used the structure and style of other testing files as a basis for our own.

Although it is not known exactly if other groups are currently investigating these probabilistic program transformations in the same way as we are, probabilistic programming has been around for some time and so there are many other active projects in this domain.

One pattern for designing probabilistic languages is to embed them in a host language. For example, the language of Hakaru is embedded in Haskell. One such other example of this pattern is WebPPL [2], which is a small probabilistic programming language embedded in JavaScript. There is plenty of future work to be done in terms of making efforts to implement PPLs using various other languages.

4 Standard Library Development

As discussed earlier, one of our major contributions has been the addition of over 60 new probabilistic models. At the onset of the project, the Hakaru language offered a small set of simple, primitive distributions. We have greatly expanded this library of distributions so that more complex models can be easily created.

Whenever possible, we have implemented distributions as transformations on pre-existing models, starting with the primitive distributions that are native to Hakaru (`normal`, `categorical`, etc.). We have used the UDR (Univariate Distribution Relationship) [3] to guide the development of the standard library, as it reveals many useful relationships bewteen relevant distributions (see Figure 2). In the case of multiple possible implementations for a distribution, we have opted for the implementation which adopts the shortest path from a primitive distribution on the UDR.

In the UDR, an arrow from distribution A to distribution B shows the transformation that must be applied to A to result in B. So, new distributions are created by combining and transforming what has already been built.

Hakaru functions representing probabilistic models (i.e. functions with a `measure(<type>)` return type) cannot be used as operands in normal mathematical operations. This is to say we cannot transform models directly (i.e. we cant do algebraic transformation on the PDF). The only operator we can use on a model is bind (`<~`), to pull a sample from it. However, we are able to transform samples however we like. Therefore, we are interested in implementing transformations of the following form.

$$\boxed{R(p,q) \Rightarrow X \sim A(p) \Rightarrow f(X) \sim B(q)}$$

In the equation above, p is a set of values parameterizing the distribution A, q is a set of values parameterizing the distribution B, $R(p,\ q)$ is a set of relationships between p and q that must be satisfied, and f is a function that applies a transformation to a sample. We can expand this definition to include transformations defined in terms of an aggregation of multiple

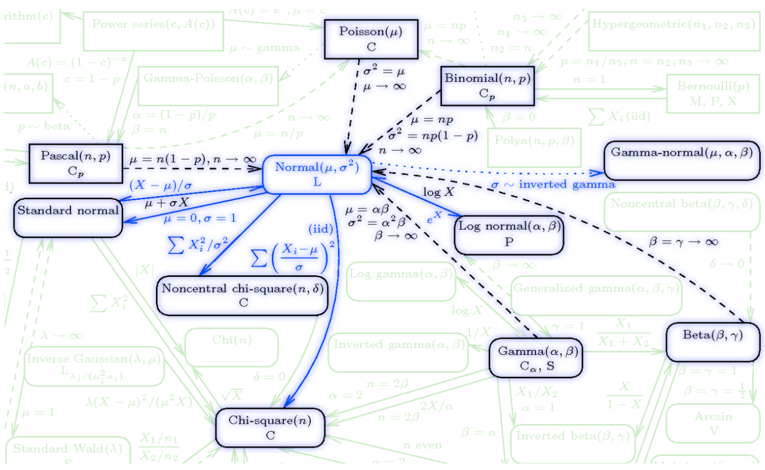
Poisson(μ) C
Binomial(n,p) C$_p$
Pascal(n,p) C$_p$
Normal(μ,σ^2) L
Standard normal
Gamma-normal(μ,α,β)
Log normal(α,β) P
Noncentral chi-square(n,δ) C
Gamma(α,β) C$_\alpha$, S
Beta(β,γ)
Chi-square(n) C
$\sigma^2=\mu$, $\mu\to\infty$
$\mu=np$, $\sigma^2=np(1-p)$, $n\to\infty$
$\mu=n(1-p)$, $n\to\infty$
$(X-\mu)/\sigma$
$\mu+\sigma X$
$\mu=0,\sigma=1$
$\sigma\sim$ inverted gamma
log X
e^X
$\mu=\alpha\beta$, $\sigma^2=\alpha^2\beta$, $\beta\to\infty$
$\beta=\gamma\to\infty$
(iid)
$\sum X_i^2/\sigma^2$
$\sum\left(\frac{X_i-\mu}{\sigma}\right)^2$

Figure 2: A snapshot of the UDR shows how the normal distribution can be transformed into a multitude of other distributions.

independent samples. For example, the standard chi-square distribution is defined as the sum of the squares of n standard normal random variables.

Hakaru also lends itself very well to Bayesian transformations where we want to pull a sample from a distribution which is parameterized with a value sampled from another distribution. These transformations take the following form. Here, we are saying that if X is parameterized by p, then the distribution C, which is parameterized by p and q, is equal to another distribution, Y, which is parameterized by q and X.

$$\boxed{X \sim A(p) \Rightarrow Y \sim B(q, X) = C(p, q)}$$

Some distributions on the UDR are unreachable from Hakaru's primitive distributions using transformations like the 2 described above. In these cases, we have to implement the model in terms of the PMF for a discrete distribution or in terms of the PDF for a continuous distribution.

5 Testing

5.1 Test Case Selection

Recall that when developing the standard library, the UDR chart often implied multiple possible implementations for a given distribution. Naturally, we would expect alternative implementations for the same distribution to result in equivalent probabilistic models. This line of thought is the basis for the kinds of test cases we have implemented. More generally, we have the following hypothesis.

Assume we know a proven relationship between 2 statistical distributions, A and B, which allows us to transform A and B into distributions that are equivalent to each other. Further assume this transformation takes on one of the forms discussed in the Standard Library Development section. We hypothesize that by applying the appropriate transformations to implementations of A and B, we can create two Hakaru programs whose `hk-maple` outputs will be equivalent to each other. Test cases that prove our hypothesis true indicate the validity of the Hakaru language implementation. Test cases that prove our hypothesis false indicate an underlying bug in the language definition which is to be passed back to the language developers.

Because of the vast number of possible relations to test between all of the new distributions implemented, the scope of testing has been limited to just focus on a few distributions. Namely, the Chi-Squared [4], Exponential [5], Erlang [6], Cauchy [7] and Rayleigh [8] distributions.

Our team has been quite successful in contributing a lot of new knowledge for the language developers in the form of test results. We have contributed over 30 new testing files (written in the style explained above) that test pairs of Hakaru program to see if they simplify down to equivalent forms. All of these are valuable knowledge for the language developers to learn so that further improvements can be made.

5.2 Results

The full results of the tests can be found in Appendix A. It gives the name of each test, the associated files, the result and the test log in the event that the test failed.

In most of the test cases that were run, the result was a failure.

6 Interpretation

In many of these failures, the test failed because the expected output of the two program transformations are not equivalant. However, a few of these failed for different reasons. In one of these cases, the two Hakaru programs submitted were not of the same type, meaning they represented probability distributions for different data types. Two programs cannot be considered equivalent if they are different return types and so this is an automatic failure. Another failure was due in part to a language feature that is currently under construction. Earlier on in the standard library development, a `tan` function was added to the language to help implement other distributions. However, one detail that was overlooked was to add `tan` to the Haskell module that was respondible for pretty printing Hakaru code that was being run through maple. So, the `tan` method that was used in one of the tests failed because an error was raised when the output was being printed.

There were a few test cases that passed. For most of these, the transformation that they were testing were what could be considered trivial, meaning the transformation is exactly how the distribution was implemented in the first place, or that the transformation itself is very simple. There were a few test cases that actually did seem to make a the correct simplification, which woud be considered a real success.

Unfortunately, we were not able to learn enough about the nature of the testing suite and the `hk-maple` algorithms. So, it was not entirely clear what each test case was checking and so this could have contributed to a lot of the failures. In general though, a lot of the failures are likely due to one of two reasons. First, there is an error in either the implementation of the distribution or in the design of the test case written by the developer. Some distributions might have bugs still in their implementaiton. As well, the actual test cases written might not actually represent the correct transformation. The second scenario could be that the test case is actually valid, however the Maple software has not been taught about this type of simplification and so it could not possibly pass.

7 Conclusion

This project was aimed at increasing language accessibility and testing the validity of Hakaru. A lot of ground was covered in both of these aspects in this project. In particular, starting with only a handful of simple distributions, we have introduced over 60 new probabilistic models using the stategy of jumping between spots on the UDR and taking advantage of some distributions PDF/PMF. Over 30 new test cases have been added to the Hakaru testing suite that investigate known relationships between these distributions.

Although a lot of ground has been covered in expanding Hakaru, compared to other popular languages, it is still under development and has a lot of room for improvement. Features such as error handlind and import statements can be added to the language, multivariate distributions could be incorporated, and there is always room for more tests.

References

[1] P. Narayanan, J. Carette, W. Romano, C. Shan, and R. Zinkov, "Probabilistic inference by program transformation in hakaru (system description)," in *International Symposium on Functional and Logic Programming - 13th International Symposium, FLOPS 2016, Kochi, Japan, March 4-6, 2016, Proceedings*, pp. 62–79, Springer, 2016. `http://dx.doi.org/10.1007/978-3-319-29604-3_5`.

[2] N. D. Goodman and A. Stuhlmüller, "The design and implementation of probabilistic programming languages," 2014. `http://dippl.org`.

[3] L. Leemis, "Univariate distribution relationships." `http://www.math.wm.edu/~leemis/chart/UDR/UDR.html`.

[4] "Chi-squared distribution," 2018. `https://en.wikipedia.org/wiki/Chi-squared_distribution`.

[5] "Exponential distribution," 2018. `https://en.wikipedia.org/wiki/Exponential_distribution`.

[6] "Erlang distribution," 2018. `https://en.wikipedia.org/wiki/Erlang_distribution`.

[7] "Cauchy distribution," 2018. `https://en.wikipedia.org/wiki/Cauchy_distribution`.

[8] "Rayleigh distribution," 2018. `https://en.wikipedia.org/wiki/Rayleigh_distribution`.

A Test Result Summary

A.1 `testExponentialRelations`

Test Name: `t_exponential_to_laplace`

Expected File: `t_exponential_to_laplace.expected.hk`

```
# If X ~ Exp(   ) and Y ~ Exp(   ) then X/        Y/    ~ Laplace(1, 1).

#Laplace(1,1)
X <~ gamma(1/1, 1/1)
Y <~ gamma(1/1, 1/1)
return X + Y * (-1/1)
```

0 File: `t_exponential_to_laplace.0.hk`

```
# If X ~ Exp(   ) and Y ~ Exp(   ) then X/        Y/    ~ Laplace(1, 1).

def exponential(alpha prob):
        gamma(1/1, alpha)

def exponentialDiff(alpha prob, beta prob):
        X <~ exponential(alpha)
        Y <~ exponential(beta)
        return X/alpha - Y/beta


exponentialDiff(5/1,1/3)
```

Result: failed

haskell/Tests/TestTools.hs:130
expected:
X <∼ gamma(1/1, 1/1)
Y <∼ gamma(1/1, 1/1)
return prob2real(X) + prob2real(Y) * (-1/1)
but got:
X5 <∼ gamma(1/1, 5/1)
Y3 <∼ gamma(1/1, 1/3)
return prob2real(X5) * (+1/5) + prob2real(Y3) * (-3/1)

Test Name: `t_exponential_scale_closure`

Expected File: `t_exponential_scale_closure.expected.hk`

```
#If X ~ Exp(   ) then kX ~ Exp(   /k).

#Equivalent: If X ~ Exp(   ) then kX ~ Exp( k   )

gamma(1/1,3/2)
```

0 File: `t_exponential_scale_closure.0.hk`

```
#If X ~ Exp(  ) then kX ~ Exp(  /k).

#Equivalent: If X ~ Exp(  ) then kX ~ Exp(k  )

def exponential(alpha prob):
        gamma(1/1, alpha)

def exponentialScaled(alpha prob, k prob):
        X <~ exponential(alpha)
        return k*X

exponentialScaled(1/2,3/1)
```

Result: failed

haskell/Tests/TestTools.hs:130
expected:
gamma(1/1, 3/2)
but got:
X3 <∼ gamma(1/1, 1/2)
return X3 * (3/1)

Test Name: `t_exponential_to_pareto`
Expected File: `t_exponential_to_pareto.expected.hk`

```
# If X ~ Exponential(  ) then k*exp(X) ~ Pareto(  , k)

#pareto(3/2,2/3)
X <~ uniform(+0/1,+1/1)
return 3/2 * (real2prob(X) ** (-3/2))
```

0 File: `t_exponential_to_pareto.0.hk`

```
# If X ~ Exponential(  ) then k*exp(X) ~ Pareto(  , k)

def exponential(alpha prob):
        gamma(1/1, alpha)

def expToPareto(lambda prob, kappa prob):
        X <~ exponential(1/lambda)
        return kappa*exp(X)


expToPareto(3/2,2/3)
```

Result: failed

haskell/Tests/TestTools.hs:130
expected:
X <∼ uniform(+0/1, +1/1)
return 3/2 * real2prob(X) ** (-3/2)
but got:
X3 <∼ gamma(1/1, 2/3)
return exp(prob2real(X3)) * (2/3)

Test Name: `t_pareto_to_exponential`
Expected File: `t_pareto_to_exponential.expected.hk`

```
#If X ~ Pareto(  , 1) then log(X) ~ Exp(  ).

#exponential(1/1)
gamma(1/1,1/1)
```

0 File: `t_pareto_to_exponential.0.hk`

```
#If X ~ Pareto(  , 1) then log(X) ~ Exp(  ).

def pareto(lambda prob, kappa prob):
        X <~ uniform(0,1)
        return lambda / (real2prob(X) ** (1/kappa))

def paretoToExponential(alpha prob):
        X <~ pareto(1/alpha,1/1)
        return real2prob(log(X))

paretoToExponential(1/1)
```

Result: failed

haskell/Tests/TestTools.hs:130
expected:
gamma(1/1, 1/1)
but got:
X3 <∼ uniform(+0/1, +1/1)
return real2prob(log(real2prob(X3)) * (-1/1))

Test Name: `t_exp_erlang_to_pareto`
Expected File: `t_exp_erlang_to_pareto.expected.hk`

```
# If X ~ Exp(  ) and Y ~ Erlang(n,   ) then: X/Y + 1 ~ Pareto(n,1)

#pareto(1,1)
X <~ uniform(+0/1,+1/1)
return real2prob(1/X)
```

0 File: `t_exp_erlang_to_pareto.0.hk`

```
# If X ~ Exp(  ) and Y ~ Erlang(n,   ) then: X/Y + 1 ~ Pareto(n,1)

def exponential(alpha prob):
        gamma(1/1, alpha)

def erlang(shape nat, scale prob) measure(prob):
        gamma(nat2prob(shape), scale)

def expErlang2pareto(n nat, alpha prob):
        X <~ exponential(alpha)
        Y <~ erlang(n, alpha)
        return X/Y + 1


expErlang2pareto(1,2/1)
```

Result: failed

haskell/Tests/TestTools.hs:130
expected:
X <∼ uniform(+0/1, +1/1)
return real2prob(1/ X)
but got:
X5 <∼ gamma(1/1, 2/1)
Y3 <∼ gamma(1/1, 2/1)
return (X5 + Y3) / Y3

Test Name: `t_exponential_sum_rates`
Expected File: `t_exponential_sum_rates.expected.hk`

```
# If X ~ Exp(   ) and Y ~ Exp(   )
#        then min(X, Y) ~ Exp(    +    ).

# Rewrite for scale:
#           = 1/    and     = 1/
#           +     = (    +    )/
#        If X ~ Exp(   ) and Y ~ Exp(   )
#        then min(X, Y) ~ Exp(     /(    +    )).

#exponential(1*2/(1+2))
gamma(1/1, 2/3)
```

0 File: `t_exponential_sum_rates.0.hk`

```
# If X ~ Exp(   ) and Y ~ Exp(   )
#        then min(X, Y) ~ Exp(    +    ).

# Rewrite for scale:
#           = 1/    and     = 1/
#           +     = (    +    )/
#        If X ~ Exp(   ) and Y ~ Exp(   )
#        then min(X, Y) ~ Exp(     /(    +    )).

def exponential(alpha prob):
        gamma(1/1, alpha)

def expSumRates(alpha prob, beta prob):
        X <~ exponential(alpha)
        Y <~ exponential(beta)
        return min(X,Y)

expSumRates(1/1,2/1)
```

Result: failed

haskell/Tests/TestTools.hs:130
expected:
gamma(1/1, 2/3)
but got:
X5 <∼ gamma(1/1, 1/1)
Y3 <∼ gamma(1/1, 2/1)

return min(X5, Y3)

Test Name: `t_exponential_to_beta`
Expected File: `t_exponential_to_beta.expected.hk`

```
# If X ~ Exp(  ) then e^  X  ~ Beta(  , 1).

beta(1/2,1/1)
```

0 File: `t_exponential_to_beta.0.hk`

```
# If X ~ Exp(  ) then e^  X  ~ Beta(1/  , 1).

def exponential(alpha prob):
        gamma(1/1, alpha)

def exp2beta(alpha prob):
        X <~ exponential(alpha)
        return exp(-X)

exp2beta(2/1)
```

Result: failed

haskell/Tests/TestTools.hs:130
expected:
beta(1/2, 1/1)
but got:
X3 <∼ gamma(1/1, 2/1)
return exp(prob2real(X3) * (-1/1))

Test Name: `t_exponential_to_stdChiSq`
Expected File: `t_exponential_to_stdChiSq.expected.hk`

```
#If X ~ Exp(2) then X        (2)^2

# Chi Squared distribution with n independant and identical distributions (iid).
def chiSq_iid(n nat, mean real, stdev prob):
        q <~ plate _ of n: normal(mean,stdev)
        return summate i from 0 to size(q):
                ((q[i]-mean)/stdev)^2

# Standard Chi Squared distribution is defined in terms of n independant
# standard normal distributions
def standardChiSq(n nat):
        chiSq_iid(n,0,1)

standardChiSq(2)
```

0 File: `t_exponential_to_stdChiSq.0.hk`

```
#If X ~ Exp(2) then X        (2)^2

def exponential(alpha prob):
        X <~ uniform(0,1)
        return -1 * alpha * log(real2prob(X))

exponential(2)
```

Result: passed

Test Name: `t_exponential_sum_to_erlang`
Expected File: `t_exponential_sum_to_erlang.expected.hk`

```
# If Xi ~ exponential( ) then X1+...+Xk ~ erlang(k, )

#erlang(2,2/1)
gamma(2/1, 2/1)
```

0 File: `t_exponential_sum_to_erlang.0.hk`

```
# If Xi ~ exponential( ) then X1+...+Xk ~ erlang(k, )

def exponential(alpha prob):
        gamma(1/1, alpha)

def exponentialSum(k nat, alpha prob):
        iid <~ plate _ of k: exponential(alpha)
        return summate i from 0 to size(iid): iid[i]


exponentialSum(2,2/1)
```

Result: failed

```
haskell/Tests/TestTools.hs:130
expected:
gamma(2/1, 2/1)
but got:
iid307 <~ gamma(1/1, 2/1)
iid315 <~ gamma(1/1, 2/1)
return iid307 + iid315
```

A.2 `testErlangRelations`

Test Name: `t_exponential_to_erlang`
Expected File: `t_exponential_to_erlang.expected.hk`

```
fn k nat:
  fn lambda prob:
    X <~ gamma(nat2prob(k),lambda)
    return(prob2real(X))
```

0 File: `t_exponential_to_erlang.0.hk`

```
def exponential(alpha prob):
  X <~ uniform(0,1)
  return -1 * alpha * log(real2prob(X))

fn k nat:
  fn lambda prob:
    X <~ plate _ of k: exponential(lambda)
    return(summate i from 0 to k: X[i])
```

Result: failed

haskell/Tests/TestTools.hs:130
expected:
fn k nat:
fn lambda prob:
X <~ gamma(nat2prob(k), lambda)
return prob2real(X)
but got:
fn k nat:
fn lambda prob:
X7 <~ plate _ of k: uniform(+0/1, +1/1)
return (summate i from 0 to k: log(real2prob(X7[i])))
* prob2real(lambda)
* (-1/1)

Test Name: `t_erlang_to_pareto`
Expected File: `t_erlang_to_pareto.expected.hk`

```
fn n nat:
        X <~ uniform(0,1)
        return (1 / (real2prob(X) ** (1/n))) #this is our implimentatio of pareto
```

0 File: `t_erlang_to_pareto.0.hk`

```
def exponential(alpha prob):
        X <~ uniform(0,1)
        return (-1 * alpha * log(real2prob(X)))

def erlang(shape nat, scale prob) measure(prob):
        gamma(nat2prob(shape), scale)

fn n nat:
        fn lambda prob:
                U <~ exponential(lambda)
                V <~ erlang(n,lambda)
                return real2prob(U/V+1)
```

Result: failed

haskell/Tests/TestTools.hs:130
expected:
fn n nat:
X <~ uniform(nat2real(0), nat2real(1))
return 1/ real2prob(X) ** prob2real(1/ nat2prob(n))
but got:
fn n nat:
X5 <~ uniform(+0/1, +1/1)
return real2prob(1/ X5) * nat2prob(n)

haskell/Tests/TestTools.hs:176
Files don't have same type (fn n nat:
X <∼ uniform(0,1)
return (1 / (real2prob(X) ** (1/n))) #this is our implimentatio of pareto
:: nat -¿ measure(prob), def exponential(alpha prob):
X <∼ uniform(0,1)
return (-1 * alpha * log(real2prob(X)))
def erlang(shape nat, scale prob) measure(prob):
gamma(nat2prob(shape), scale)
fn n nat:
fn lambda prob:
U <∼ exponential(lambda)
V <∼ erlang(n,lambda)
return real2prob(U/V+1)
:: nat -¿ prob -¿ measure(prob)

Test Name: `t_erlang_to_erlang_1`
Expected File: `t_erlang_to_erlang_1.expected.hk`

```
fn shape1 nat:
  fn shape2 nat:
    fn scale prob:
      gamma(nat2prob(shape1)+nat2prob(shape2),scale)
```

0 File: `t_erlang_to_erlang_1.0.hk`

```
def erlang(shape nat, scale prob) measure(prob):
  gamma(nat2prob(shape), scale)

fn shape1 nat:
  fn shape2 nat:
    fn scale prob:
      X <~ erlang(shape1,scale)
      Y <~ erlang(shape2,scale)
      return(X+Y)
```

Result: passed

Test Name: `t_erlang_to_stdChiSq`
Expected File: `t_erlang_to_stdChiSq.expected.hk`

```
fn k nat:
        q <~ plate _ of 2*k: normal(0,1)
        return summate i from 0 to size(q): (q[i])^2   # our implementation of
             stdChiSq
```

0 File: `t_erlang_to_stdChiSq.0.hk`

```
def erlang(shape nat, scale prob) measure(prob):
        gamma(nat2prob(shape), scale)

fn k nat:
        fn lambda prob:
                X <~ erlang(k,lambda)
                return 2*lambda*X
```

Result: failed

haskell/Tests/TestTools.hs:130
expected:
fn k nat:
q <∼ plate _ of 2 * k: normal(nat2real(0), nat2prob(1))
return summate i from 0 to size(q): q[i] ** 2
but got:
fn k nat:
q5 <∼ plate _ of k * 2: normal(+0/1, 1/1)
return summate i from 0 to k * 2: q5[i] ** 2

haskell/Tests/TestTools.hs:176
Files don't have same type (fn k nat:
q <∼ plate _ of 2*k: normal(0,1)
return summate i from 0 to size(q): (q[i])**2 # our implementation of stdChiSq :: nat -¿ measure(real), def erlang(shape nat, scale prob) measure(prob):
gamma(nat2prob(shape), scale)
fn k nat:
fn lambda prob:
X <∼ erlang(k,lambda)
return 2*lambda*X

:: nat -¿ prob -¿ measure(prob)

Test Name: `t_erlang_to_erlang`
Expected File: `t_erlang_to_erlang.expected.hk`

```
fn scale prob:
  fn shape nat:
    fn a prob:
      gamma(nat2prob(shape)/a, scale)
```

0 File: `t_erlang_to_erlang.0.hk`

```
def erlang(shape nat, scale prob) measure(prob):
  gamma(nat2prob(shape), scale)
fn scale prob:
  fn shape nat:
    fn a prob:
      x <~ erlang(shape, scale)
      return(x*a)
```

Result: failed

haskell/Tests/TestTools.hs:130
expected:

fn shape1 nat:
fn shape2 nat:
fn scale prob: gamma(nat2prob(shape1) + nat2prob(shape2), scale)
but got:
fn shape1 nat:
fn shape2 nat:
fn scale prob:
Xb <∼ gamma(nat2prob(shape1), scale)
Y9 <∼ gamma(nat2prob(shape2), scale)
return Xb + Y9

A.3 `testStdChiSqRelations`

Test Name: `t_stdChiSq_superposition`
Expected File: `t_stdChiSq_superposition.expected.hk`

```
def chiSq_iid(n nat, mean real, stdev prob):
        q <~ plate _ of n: normal(mean,stdev)
        return summate i from 0 to size(q):
                ((q[i]-mean)/stdev)^2

# Chi distribution with n independant and identical distributions (iid).
def chi_iid(n nat, mean real, stdev prob):
        q <~ chiSq_iid(n,mean,stdev)
        return sqrt(real2prob(q))

# Standard Chi Squared distribution is defined in terms of n independant
# standard normal distributions
def standardChiSq(n nat):
        chiSq_iid(n,0,1)


x <~ standardChiSq(3)

return x
```

0 File: `t_stdChiSq_superposition.0.hk`

```
def chiSq_iid(n nat, mean real, stdev prob):
        q <~ plate _ of n: normal(mean,stdev)
        return summate i from 0 to size(q):
                ((q[i]-mean)/stdev)^2

# Chi distribution with n independant and identical distributions (iid).
def chi_iid(n nat, mean real, stdev prob):
        q <~ chiSq_iid(n,mean,stdev)
        return sqrt(real2prob(q))

# Standard Chi Squared distribution is defined in terms of n independant
# standard normal distributions
def standardChiSq(n nat):
        chiSq_iid(n,0,1)


x1 <~ standardChiSq(1)
x2 <~ standardChiSq(2)

return x1 + x2
```

Result: failed

```
haskell/Tests/TestTools.hs:130
expected:
chiSq_iid = fn n nat:
fn mean real:
fn stdev prob:
q <~ plate _ of n: normal(mean, stdev)
return summate i from 0 to size(q):
((q[i] - mean) * prob2real(1/ stdev)) ** 2
chi_iid = fn n nat:
fn mean real:
fn stdev prob:
q <~ chiSq_iid(n, mean, stdev)
return sqrt(real2prob(q))
standardChiSq = fn n nat: chiSq_iid(n, nat2real(0), nat2prob(1))
x <~ standardChiSq(3)
return x
but got:
q309 <~ normal(+0/1, 1/1)
q317 <~ normal(+0/1, 1/1)
q325 <~ normal(+0/1, 1/1)
return q309 ** 2 + q317 ** 2 + q325 ** 2
```

```
haskell/Tests/TestTools.hs:130
expected:
chiSq_iid = fn n nat:
fn mean real:
fn stdev prob:
q <~ plate _ of n: normal(mean, stdev)
return summate i from 0 to size(q):
((q[i] - mean) * prob2real(1/ stdev)) ** 2
chi_iid = fn n nat:
fn mean real:
fn stdev prob:
q <~ chiSq_iid(n, mean, stdev)
return sqrt(real2prob(q))
standardChiSq = fn n nat: chiSq_iid(n, nat2real(0), nat2prob(1))
x <~ standardChiSq(3)
return x
but got:
q90b <~ normal(+0/1, 1/1)
q307 <~ normal(+0/1, 1/1)
q315 <~ normal(+0/1, 1/1)
return q307 ** 2 + q315 ** 2 + q90b ** 2
```

Test Name: `t_stdChiSq_to_gamma`
Expected File: `t_stdChiSq_to_gamma.expected.hk`

```
# gamma(v/2,2c) = gamma(1/2,1)
# v = 1
# c = 1/2

gamma(1/2,1/1)
```

0 File: `t_stdChiSq_to_gamma.0.hk`

```
# X ~ standardChiSq(v)
# c*X ~ gamma(v/2,2c) = gamma(0.5,1)
# v = 1
# c = 0.5

# This function takes 2 arrays of size N, containing the means and standard
    deviations
# of the N normal distributions.
def chiSq(means array(real), stdevs array(prob) ):
        #TODO: error check sizes of means==stdevs
        q <~ plate _ of size(means): normal(means[_],stdevs[_])
        return summate i from 0 to size(q):
                ((q[i]-means[i])/stdevs[i])^2
# Chi Squared distribution with n independant and identical distributions (iid).
def chiSq_iid(n nat, mean real, stdev prob):
        q <~ plate _ of n: normal(mean,stdev)
        return summate i from 0 to size(q):
                ((q[i]-mean)/stdev)^2
# Standard Chi Squared distribution is defined in terms of n independant
# standard normal distributions
def standardChiSq(n nat):
        chiSq_iid(n,0,1)

chiSq2gamma = fn v nat:
                                fn c real:
                                        X <~ standardChiSq(v)
                                        return real2prob(c*X)

chiSq2gamma(1,1/2)
```

Result: failed

haskell/Tests/TestTools.hs:130
expected:
gamma(1/2, 1/1)
but got:
q305 <~ normal(+0/1, 1/1)
return real2prob(q305 ** 2) * (1/2)

Test Name: `t_stdChiSq_to_exponential`
Expected File: `t_stdChiSq_to_exponential.expected.hk`

```
def exponential(alpha prob):
        X <~ uniform(+0/1,+1/1)
        return -alpha * log(real2prob(X))

exponential(1/2)
```

0 File: `t_stdChiSq_to_exponential.0.hk`

```
# This function takes 2 arrays of size N, containing the means and standard
    deviations
# of the N normal distributions.
def chiSq(means array(real), stdevs array(prob) ):
        #TODO: error check sizes of means==stdevs
        q <~ plate _ of size(means): normal(means[_],stdevs[_])
        return summate i from 0 to size(q):
                ((q[i]-means[i])/stdevs[i])^2
# Chi Squared distribution with n independant and identical distributions (iid).
def chiSq_iid(n nat, mean real, stdev prob):
        q <~ plate _ of n: normal(mean,stdev)
        return summate i from 0 to size(q):
                ((q[i]-mean)/stdev)^2
# Standard Chi Squared distribution is defined in terms of n independant
# standard normal distributions
def standardChiSq(n nat):
        chiSq_iid(n,0,1)

standardChiSq(2)
```

Result: failed

haskell/Tests/TestTools.hs:130
expected:
exponential = fn alpha prob:
X <~ uniform(+0/1, +1/1)
return -prob2real(alpha) * log(real2prob(X))
exponential(1/2)
but got:
X3 <~ uniform(+0/1, +1/1)
return log(real2prob(X3)) * (-1/2)

haskell/Tests/TestTools.hs:130
expected:
exponential = fn alpha prob:
X <~ uniform(+0/1, +1/1)
return -prob2real(alpha) * log(real2prob(X))
exponential(1/2)
but got:
q307 <~ normal(+0/1, 1/1)
q315 <~ normal(+0/1, 1/1)
return q307 ** 2 + q315 ** 2

Test Name: `t_rayleigh_to_stdChiSq`
Expected File: `t_rayleigh_to_stdChiSq.expected.hk`

```
# Chi Squared distribution with n independant and identical distributions (iid).
def chiSq_iid(n nat, mean real, stdev prob):
        q <~ plate _ of n: normal(mean,stdev)
        return summate i from 0 to size(q):
                ((q[i]-mean)/stdev)^2

# Standard Chi Squared distribution is defined in terms of n independant
# standard normal distributions
def standardChiSq(n nat):
        chiSq_iid(n,0,1)

standardChiSq(2)
```

0 File: `t_rayleigh_to_stdChiSq.0.hk`

```
# Test case for the relation:
#       If R ~ rayleigh(2) then R^2 ~ stdChiSq(2)

def exponential(alpha prob):
        X <~ uniform(0,1)
        return -1 * alpha * log(real2prob(X))

def rayleigh(alpha prob):
        X <~ exponential(alpha)
        return sqrt(real2prob(X))


def Rayleigh_to_ChiSq():
        x <~ rayleigh(2)
        return prob2real(x ** 2)

Rayleigh_to_ChiSq()
```

Result: failed

haskell/Tests/TestTools.hs:130
expected:
chiSq_iid = fn n nat:
fn mean real:
fn stdev prob:
q <~ plate _ of n: normal(mean, stdev)
return summate i from 0 to size(q):
((q[i] - mean) * prob2real(1/ stdev)) ** 2
standardChiSq = fn n nat: chiSq_iid(n, nat2real(0), nat2prob(1))
standardChiSq(2)
but got:
q307 <~ normal(+0/1, 1/1)
q315 <~ normal(+0/1, 1/1)
return q307 ** 2 + q315 ** 2

haskell/Tests/TestTools.hs:130
expected:
chiSq_iid = fn n nat:
fn mean real:
fn stdev prob:
q <~ plate _ of n: normal(mean, stdev)
return summate i from 0 to size(q):
((q[i] - mean) * prob2real(1/ stdev)) ** 2
standardChiSq = fn n nat: chiSq_iid(n, nat2real(0), nat2prob(1))
standardChiSq(2)
but got:
X3 <~ uniform(+0/1, +1/1)
return log(real2prob(X3)) * (-2/1)

Test Name: `t_stdChiSq_to_beta`

Expected File: `t_stdChiSq_to_beta.expected.hk`

```
beta(1/1, 2/1)
```

0 File: `t_stdChiSq_to_beta.0.hk`

```
# Chi Squared distribution with n independant and identical distributions (iid).
def chiSq_iid(n nat, mean real, stdev prob):
        q <~ plate _ of n: normal(mean,stdev)
        return summate i from 0 to size(q):
                ((q[i]-mean)/stdev)^2

# Chi distribution with n independant and identical distributions (iid).
def chi_iid(n nat, mean real, stdev prob):
        q <~ chiSq_iid(n,mean,stdev)
        return sqrt(real2prob(q))

# Standard Chi Squared distribution is defined in terms of n independant
# standard normal distributions
def standardChiSq(n nat):
        chiSq_iid(n,0,1)

x <~ standardChiSq(2)
y <~ standardChiSq(4)
return real2prob(x / (x + y))
```

Result: failed

haskell/Tests/TestTools.hs:130
expected:
beta(1/1, 2/1)
but got:
qd0h <~ normal(+0/1, 1/1)
qd1f <~ normal(+0/1, 1/1)
q30b <~ normal(+0/1, 1/1)
q319 <~ normal(+0/1, 1/1)
q327 <~ normal(+0/1, 1/1)
q335 <~ normal(+0/1, 1/1)
return real2prob
((qd0h ** 2 + qd1f ** 2)
/ (q30b ** 2
+ q319 ** 2
+ q327 ** 2
+ q335 ** 2
+ qd0h ** 2
+ qd1f ** 2))

Test Name: `t_laplace_to_chiSq`
Expected File: `t_laplace_to_chiSq.expected.hk`

```
# stdChiSq(2)
X1 <~ normal(+0/1, 1/1)
X2 <~ normal(+0/1, 1/1)
return real2prob(X1^2 + X2^2)
```

0 File: `t_laplace_to_chiSq.0.hk`

```
def exponential(alpha prob):
        gamma(1/1, alpha)

def Laplace(alpha prob, beta prob):
        X <~ exponential(alpha)
        Y <~ exponential(beta)
        return X - Y

X <~ Laplace (1,1)
return (2 * abs(X-1))/ 1
```

Result: passed

Test Name: `t_uniform_to_stdChiSq`
Expected File: `t_uniform_to_stdChiSq.expected.hk`

```
# Chi Squared distribution with n independant and identical distributions (iid).
def chiSq_iid(n nat, mean real, stdev prob):
        q <~ plate _ of n: normal(mean,stdev)
        return summate i from 0 to size(q):
                ((q[i]-mean)/stdev)^2

# Chi distribution with n independant and identical distributions (iid).
def chi_iid(n nat, mean real, stdev prob):
        q <~ chiSq_iid(n,mean,stdev)
        return sqrt(real2prob(q))

# Standard Chi Squared distribution is defined in terms of n independant
# standard normal distributions
def standardChiSq(n nat):
        chiSq_iid(n,0,1)

standardChiSq(2)
```

0 File: `t_uniform_to_stdChiSq.0.hk`

```
x <~ uniform(0, 1)
return -2 * log(real2prob(x))
```

Result: failed

haskell/Tests/TestTools.hs:130
expected:
chiSq_iid = fn n nat:
fn mean real:
fn stdev prob:
q <~ plate _ of n: normal(mean, stdev)
return summate i from 0 to size(q):
((q[i] - mean) * prob2real(1/ stdev)) ** 2
chi_iid = fn n nat:
fn mean real:
fn stdev prob:
q <~ chiSq_iid(n, mean, stdev)
return sqrt(real2prob(q))
standardChiSq = fn n nat: chiSq_iid(n, nat2real(0), nat2prob(1))

standardChiSq(2)
but got:
q307 <∼ normal(+0/1, 1/1)
q315 <∼ normal(+0/1, 1/1)
return q307 ** 2 + q315 ** 2

haskell/Tests/TestTools.hs:130
expected:
chiSq_iid = fn n nat:
fn mean real:
fn stdev prob:
q <∼ plate _ of n: normal(mean, stdev)
return summate i from 0 to size(q):
((q[i] - mean) * prob2real(1/ stdev)) ** 2
chi_iid = fn n nat:
fn mean real:
fn stdev prob:
q <∼ chiSq_iid(n, mean, stdev)
return sqrt(real2prob(q))
standardChiSq = fn n nat: chiSq_iid(n, nat2real(0), nat2prob(1))
standardChiSq(2)
but got:
x3 <∼ uniform(+0/1, +1/1)
return log(real2prob(x3)) * (-2/1)

A.4 `testCauchyRelations`

Test Name: `t_uniform_to_cauchy`
Expected File: `t_uniform_to_cauchy.expected.hk`

```
#cauchy(0,1)
X1 <~ normal(+0/1, 1/1)
X2 <~ normal(+0/1, 1/1)
return X1/X2
```

0 File: `t_uniform_to_cauchy.0.hk`

```
X <~ uniform(0, 1)
return tan(pi * (X - 1/2))
```

Result: passed

Test Name: `t_cauchy_add_transformation`
Expected File: `t_cauchy_add_transformation.expected.hk`

```
# cauchy(4, 6)
X1 <~ normal(+0/1, 1/1)
X2 <~ normal(+0/1, 1/1)
return (X2 * (+2/1) + X1 * (+3/1)) / X2 * (+2/1)
```

0 File: `t_cauchy_add_transformation.0.hk`

```
def stdNormal():
        p <~ normal(0, 1)
        return p

def stdCauchy():
        X1 <~ stdNormal()
        X2 <~ stdNormal()
        return X1/X2

def cauchy(a real, alpha prob):
        X <~ stdCauchy()
        return a + alpha*X

X <~ cauchy(1, 2)
Y <~ cauchy(3, 4)
return X + Y
```

Result: failed

haskell/Tests/TestTools.hs:130
expected:
X1 <∼ normal(+0/1, 1/1)
X2 <∼ normal(+0/1, 1/1)
return (X2 * (+2/1) + X1 * (+3/1)) / X2 * (+2/1)
but got:
pb <∼ normal(+0/1, 1/1)
p9 <∼ normal(+0/1, 1/1)
p7 <∼ normal(+0/1, 1/1)
p5 <∼ normal(+0/1, 1/1)
return (p5 * p9 * (+2/1) + p5 * pb + p7 * p9 * (+2/1))
/ p9
/ p5
* (+2/1)

Test Name: `t_cauchy_sub_transformation`
Expected File: `t_cauchy_sub_transformation.expected.hk`

```
# cauchy(-2, 6)
X1 <~ normal(+0/1, 1/1)
X2 <~ normal(+0/1, 1/1)
return (X2 * (-1/1) + X1 * (+3/1)) / X2 * (+2/1)
```

0 File: `t_cauchy_sub_transformation.0.hk`

```
def stdNormal():
        p <~ normal(0, 1)
        return p

def stdCauchy():
        X1 <~ stdNormal()
        X2 <~ stdNormal()
        return X1/X2

def cauchy(a real, alpha prob):
        X <~ stdCauchy()
        return a + alpha*X

X <~ cauchy(1, 2)
Y <~ cauchy(3, 4)
return X - Y
```

Result: failed

```
haskell/Tests/TestTools.hs:130
expected:
X1 <~ normal(+0/1, 1/1)
X2 <~ normal(+0/1, 1/1)
return (X2 * (-1/1) + X1 * (+3/1)) / X2 * (+2/1)
but got:
pb <~ normal(+0/1, 1/1)
p9 <~ normal(+0/1, 1/1)
p7 <~ normal(+0/1, 1/1)
p5 <~ normal(+0/1, 1/1)
return (p5 * p9 + p5 * pb * (-1/1) + p7 * p9 * (+2/1))
/ p9
/ p5
* (-2/1)
```

Test Name: `t_cauchy_reciprocal_transformation`
Expected File: `t_cauchy_reciprocal_transformation.expected.hk`

```
#cauchy(0,1/2)
X1 <~ normal(+0/1, 1/1)
X2 <~ normal(+0/1, 1/1)
return 1/X2*X1*(+1/2)
```

0 File: `t_cauchy_reciprocal_transformation.0.hk`

```
def stdNormal():
        p <~ normal(0, 1)
        return p

def stdCauchy():
        X1 <~ stdNormal()
        X2 <~ stdNormal()
        return X1/X2

def cauchy(a real, alpha prob):
        X <~ stdCauchy()
        return a + alpha*X

X <~ cauchy(0, 2)
return 1 / X
```

Result: failed

haskell/Tests/TestTools.hs:130
expected:
X1 <∼ normal(+0/1, 1/1)
X2 <∼ normal(+0/1, 1/1)
return X1 / X2 * (+1/2)
but got:
p5 <∼ normal(+0/1, 1/1)
p3 <∼ normal(+0/1, 1/1)
return 1/ p5 * p3 * (+1/2)

Test Name: `t_cauchy_linear_transformation`
Expected File: `t_cauchy_linear_transformation.expected.hk`

```
# cauchy(7, 2)
X1 <~ normal(+0/1, 1/1)
X2 <~ normal(+0/1, 1/1)
return (X1 * (+2/1) + X2 * (+7/1)) / X2
```

0 File: `t_cauchy_linear_transformation.0.hk`

```
def stdNormal():
        p <~ normal(0, 1)
        return p

def stdCauchy():
        X1 <~ stdNormal()
        X2 <~ stdNormal()
        return X1/X2

def cauchy(a real, alpha prob):
        X <~ stdCauchy()
        return a + alpha*X

X <~ cauchy(2, 1)
return 2*X + 3
```

Result: passed

Test Name: `t_cauchy_to_students_t`
Expected File: `t_cauchy_to_students_t.expected.hk`

```
#t(1)
U <~ normal(+0/1,1/1)
X <~ normal(+0/1,1/1)
return U/X
```

0 File: `t_cauchy_to_students_t.0.hk`

```
def stdNormal():
        p <~ normal(0, 1)
        return p

def stdCauchy():
        X1 <~ stdNormal()
        X2 <~ stdNormal()
        return X1/X2

def cauchy(a real, alpha prob):
        X <~ stdCauchy()
        return a + alpha*X

cauchy(0, 1)
```

Result: failed

haskell/Tests/TestTools.hs:130
expected:
U <∼ normal(+0/1, 1/1)
X <∼ normal(+0/1, 1/1)
return U * prob2real(1/ abs(X))
but got:
p5 <∼ normal(+0/1, 1/1)
p3 <∼ normal(+0/1, 1/1)
return p5 / p3

A.5 `testRayleighRelations`

Test Name: `t_exponential_to_rayleigh`
Expected File: `t_exponential_to_rayleigh.expected.hk`

```
#rayleigh(1)
X3 <~ uniform(+0/1, +1/1)
return sqrt(real2prob(log(real2prob(X3)) * (-1/1)))
```

0 File: `t_exponential_to_rayleigh.0.hk`

```
def exponential(alpha prob):
        gamma(1/1,alpha)

X <~ exponential(1)
return sqrt(X)
```

Result: failed

haskell/Tests/TestTools.hs:130
expected:
X3 <∼ uniform(+0/1, +1/1)
return sqrt(real2prob(log(real2prob(X3)) * (-1/1)))
but got:

X3 <~ gamma(1/1, 1/1)
return sqrt(X3)

Test Name: `t_rayleigh_to_gamma`
Expected File: `t_rayleigh_to_gamma.expected.hk`

```
gamma(1/1,2/1)
```

0 File: `t_rayleigh_to_gamma.0.hk`

```
def exponential(alpha prob):
        X <~ uniform(0,1)
        return -1 * alpha * log(real2prob(X))

def rayleigh(alpha prob):
        X <~ exponential(alpha)
        return sqrt(real2prob(X))

X <~ rayleigh(1)
return X ** 2
```

Result: failed

haskell/Tests/TestTools.hs:130
expected:
gamma(1/1, 2/1)
but got:
X3 <~ uniform(+0/1, +1/1)
return real2prob(log(real2prob(X3)) * (-1/1))

Test Name: `t_weibull_to_rayleigh`
Expected File: `t_weibull_to_rayleigh.expected.hk`

```
#rayleigh(1)
X <~ uniform(+0/1, +1/1)
return sqrt(real2prob(log(real2prob(X)) * (-1/1)))
```

0 File: `t_weibull_to_rayleigh.0.hk`

```
def exponential(alpha prob):
        X <~ uniform(0,1)
        return -1 * alpha * log(real2prob(X))

def weibull(alpha prob, beta prob):
        X <~ exponential(alpha)
        return real2prob(X) ** (1/beta)

X <~ weibull(1,1)
return sqrt(real2prob(X))
```

Result: passed