Formalization of Continuous Probability Distributions

Osman Hasan and Sofiène Tahar

Department of Electrical and Computer Engineering,
Concordia University, Montreal, Canada
Email: {o_hasan, tahar}@ece.concordia.ca

Technical Report

February, 2007

Abstract

In order to overcome the limitations of state-of-the-art simulation based probabilistic analysis, we propose to perform probabilistic analysis within the environment of a higher-order-logic theorem prover. The foremost requirement for conducting such analysis is the formalization of probability distributions. In this report, we present a methodology for the formalization of continuous probability distributions for which the inverse of the cumulative distribution function can be expressed in a closed mathematical form. Our methodology is primarily based on the formalization of the Standard Uniform random variable, cumulative distribution function properties and the Inverse Transform method. The report presents all this formalization using the HOL theorem prover. In order to illustrate the practical effectiveness of our methodology, the formalization of a few continuous probability distributions has also been included.

1 Introduction

Probabilistic analysis has become a tool of fundamental importance to virtually all scientists and engineers as they often have to deal with systems that exhibit significant random or unpredictable elements. The main idea behind probabilistic analysis is to model these uncertainties by random variables and then judge the performance and reliability issues based on the corresponding probabilistic properties.

Random variables are basically functions that map random events to numbers. Every random variable gives rise to a probability distribution, which contains most of the important information about this random variable. The probability distribution of a random variable can be uniquely described by its *cumulative distribution function* (CDF) which is sometimes also referred to as the probability distribution function. The CDF of a random variable R, F_R(x), represents the probability that the random variable R takes on a value that is less than or equal to a real number x

$$F_R(x) = Pr(R \leq x) \tag{1}$$

where Pr represents the probability. A distribution is called discrete if its CDF consists of a sequence of finite jumps, which means that it belongs to a random variable that can only attain values from a certain finite or countable set. Similarly, a distribution is called continuous if its CDF is continuous, which means that it belongs to a random variable that ranges over a continuous set of numbers. A continuous set of numbers, sometimes referred to as an interval, contains all real numbers between two limits. An interval can be open (a,b) corresponding to the set $\{x|a < x < b\}$, closed [a,b] which represents the set $\{x|a \leq x \leq b\}$, or half-open (a,b], [a,b).

Today, simulation is the most commonly used computer based probabilistic analysis technique. Most simulation softwares provide a programming environment for defining functions that approximate random variables for probability distributions. The random elements in a given system are modeled by these functions and the system is analyzed using computer simulation techniques [4], such as the Monte Carlo Method [21], where the main idea is to approximately answer a query on a probability distribution by analyzing a large number of samples. Due to these approximations the results can be quite unreliable at times. Another major limitation of simulation based probabilistic analysis is the enormous amount of CPU time requirement for attaining meaningful estimates. This approach generally requires hundreds of thousands of simulations to calculate the probabilistic quantities and becomes impractical when each simulation step involves extensive computations.

As an alternative to simulation techniques, we propose to use higher-order logic interactive theorem proving [9] for probabilistic analysis. Higher-order logic is a system of deduction with a precise semantics and can be used for the development of almost all classical mathematics theories. Interactive theorem proving is the field of computer science and mathematical logic concerned with computer based formal proof tools that require some sort of human assistance. We believe that probabilistic analysis can be performed by specifying the behavior of systems which exhibit randomness in higher-order logic and formally proving the intended probabilistic properties within the environment of an interactive theorem prover. Due to the inherent soundness of this approach, the probabilistic analysis carried out in this way will be precisely accurate.

The foremost criteria for implementing a formalized probabilistic analysis framework is to be able to formalize random variables in higher-order logic. Hurd's PhD thesis [15] can be considered a pioneering work in this regard as it presents a methodology for the formalization of probabilistic algorithms in the higher-order-logic (HOL) theorem prover [10]. Random variables are basically probabilistic algorithms and Hurd formalized some discrete random variables in [15]. On the other hand, Hurd's methodology cannot be used, as is, to formalize continuous random variables. In fact, to the best of our knowledge, no higher-order-logic formalization of continuous random variables exists in the literature so far.

1.1 Proposed Methodology

In this report, we propose a methodology for the formalization of continuous random variables based on Hurd's formalization framework and nonuniform random number generation methods [7]. The process of obtaining random variates with arbitrary distributions using a uniform *random number generator* (RNG) is termed as nonuniform random number generation. All computer based RNGs generate uniformly distributed numbers [18] and nonuniform random generation methods are quite commonly used in applications which call for other kinds of distributions. Random number generation has intrigued scientists for a few decades, and a lot of effort has been spent in order to obtain efficient and accurate algorithms for various continuous random variables. The proposed methodology is based on the fact that this enormous amount of research can be utilized for the formalization of continuous probability distributions in a higher-order logic theorem prover. The main advantage of this approach is that we only need to formalize one continuous random variable from scratch; i.e. the Standard Uniform random variable. The other continuous random variables can then be formalized by using the formalized Standard Uniform random variable and formalizing the corresponding nonuniform random number generation method.

Next, we utilize the above methodology to construct a framework, illustrated in Figure 1, for the formalization of continuous probability distributions for which the inverse of the CDF can be represented in a closed mathematical form. The first step is to formally specify the Standard Uniform random variable and verify its correctness by proving the corresponding CDF, *probability mass function* (PMF) and measurability properties. The next step is the formalization of the CDF and the verification of the corresponding properties. Then, we propose to formally specify the inverse function of a CDF in the HOL theorem prover. This formal specification, along with the formalization of the Standard Unform random variable and the CDF properties, can be used to formally verify the correctness of the *Inverse Transform Method* (ITM) [7], which is a well known nonuniform random generation technique for generating nonuniform random variates for continuous probability distributions for which the inverse of the CDF can be represented in a closed mathematical form. Now any continuous random variable, for which the inverse of the CDF can be represented in a closed form, can be formally specified in terms of the formalized Standard Uniform random variable and its corresponding CDF can be verified using the correctness proof of the ITM.

1.2 Report Outline

The report is organized as follows: In Section 2, we provide an overview of the HOL theorem prover and Hurd's methodology for the formalization of probabilistic algorithms in HOL.

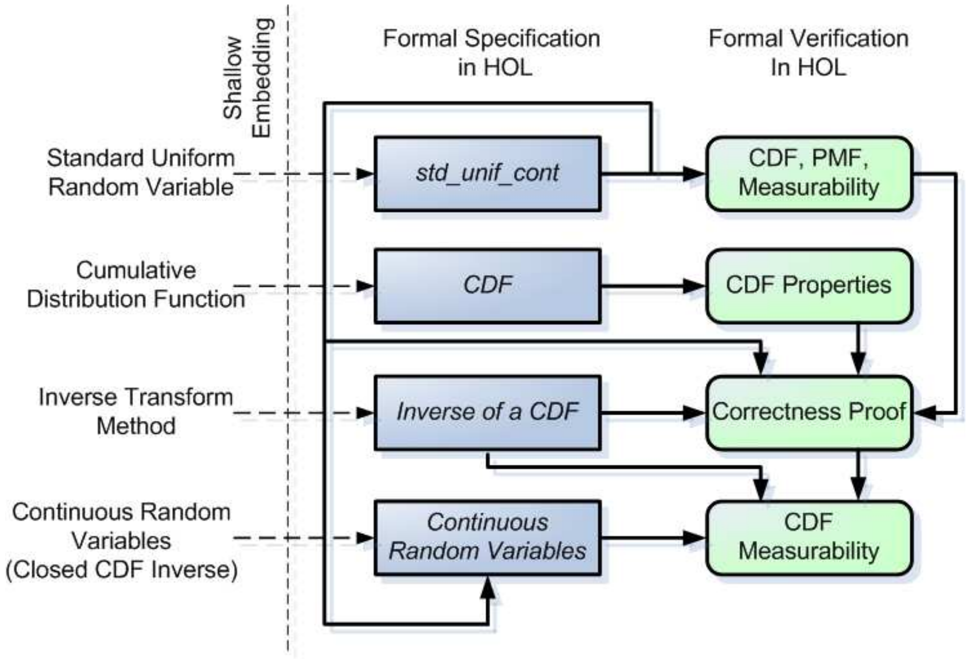
Shallow Embedding
Formal Specification in HOL
Formal Verification In HOL
Standard Uniform Random Variable
Cumulative Distribution Function
Inverse Transform Method
Continuous Random Variables (Closed CDF Inverse)
std_unif_cont
CDF
Inverse of a CDF
Continuous Random Variables
CDF, PMF, Measurability
CDF Properties
Correctness Proof
CDF Measurability

Figure 1: Proposed Formalization Framework

The next four sections of this report present the HOL formalization of the four major steps given in Figure 1, i.e, the Standard Uniform random variable, CDF, ITM and continuous probability distributions, for which the inverse of the CDF can be represented in a closed mathematical form, respectively. In Section 7, we mention some of the potential engineering applications that can be formally analyzed using our formalized continuous probability distributions. A review of the related work in the literature is given in Section 8 and we finally conclude the report in Section 9.

2 Preliminaries

In this section, we provide an overview of the HOL theorem prover and Hurd's methodology [15] for the formalization of probabilistic algorithms in HOL. The intent is to provide a brief introduction to these topics along with some notation that is going to be used in the next sections.

2.1 HOL Theorem Prover

The HOL theorem prover is an interactive theorem prover which is capable of conducting proofs in higher-order logic. It utilizes the simple type theory of Church [5] along with Hindley-Milner polymorphism [22] to implement higher-order logic. HOL has been successfully used as a verification framework for both software and hardware as well as a platform for the formalization of pure mathematics. It supports the formalization of various mathematical theories including sets, natural numbers, real numbers, measure and probability. HOL is an interactive theorem prover with access to many proof assistants and automatic proof procedures. The user interacts with a proof editor and provides it with the necessary tactics to prove goals while some of the proof steps are solved automatically by the automatic proof procedures.

In order to ensure secure theorem proving, the logic in the HOL system is represented in the strongly-typed functional programming language ML [25]. The ML abstract data types are then used to represent higher-order-logic theorems and the only way to interact with the theorem prover is by executing ML procedures that operate on values of these data types. Users can prove theorems using a natural deduction style by applying inference rules to axioms or previously generated theorems. The HOL core consists of only 5 axioms and 8 primitive inference rules, which are implemented as ML functions. Soundness is assured as every new theorem must eventually be created from the 5 axioms or any other pre-existing theorems and the 8 primitive inference rules.

We selected HOL theorem prover for the proposed formalization mainly because of its inherent soundness and ability to handle higher-order logic and in order to benefit from the inbuilt mathematical theories for measure and probability. Table 1 summarizes some frequently used HOL symbols in this report and their corresponding mathematical interpretation [10].

HOL Symbol	Standard Symbol	Meaning
$bool$	$\{\top, \bot\}$	Boolean data type
num	$\{0, 1, 2, \ldots\}$	Natural data type
$real$	All Real numbers	Real data type
$\lambda x.t$	$\lambda x.t$	Function that maps x to $t(x)$
$\sim t$	$\neg t$	Logical and Mathematical Negation
$\wedge$	*and*	Logical *and*
$\vee$	*or*	Logical *or*
SUC n	$n + 1$	Successor of a *num*
$m \ {*}{*} \ n$	m^n	*num* m raised to *num* exponent n
&	(none)	Maps type *num* to *real*
$x\ pow\ n$	x^n	*real* x raised to *num* power n
$inv\ x$	x^{-1}	Multiplicative inverse of a *real* x
$lim(\lambda n.f(n))$	$\lim\limits_{n\to\infty} f(n)$	Limit of a *real* sequence f
$\{x \vert P(x)\}$	$\{\lambda x.P(x)\}$	Set of all x that satisfy the condition P
(a, b)	a x b	A mathematical pair of two elements

Table 1: HOL Symbols

2.2 Verifying Probabilistic Algorithms in HOL

Hurd [15] proposed to formalize the probabilistic algorithms in higher-order logic by thinking of them as deterministic functions with access to an infinite Boolean sequence $\mathbb{B}^\infty$; a source of infinite random bits. These deterministic functions make random choices based on the result of popping the top most bit in the infinite Boolean sequence and may pop as many random bits as they need for their computation. When the algorithms terminate, they return the result along with the remaining portion of the infinite Boolean sequence to be used by other programs. Thus, a probabilistic algorithm which takes a parameter of type α and ranges over values of type β can be represented in HOL by the function

$$\mathcal{F} : \alpha \rightarrow B^{\infty} \rightarrow \beta \times B^{\infty}$$

For example, a $Bernoulli(\frac{1}{2})$ random variable that returns 1 or 0 with equal probability $\frac{1}{2}$ can be modeled as follows

```
⊢ bit = λs.  (if shd s then 1 else 0, stl s)
```

where s is the infinite Boolean sequence and `shd` and `stl` are the sequence equivalents of the list operation *'head'* and *'tail'*. The function *bit* accepts the infinite Boolean sequence and returns a random number, which is either 0 or 1 together with a sequence of unused Boolean sequence, which in this case is the tail of the sequence. The above methodology can be used to model most probabilistic algorithms. All probabilistic algorithms that compute a finite number of values equal to 2^n, each having a probability of the form $\frac{m}{2^n}$: where m represents the hol data type *nat* and is always less than 2^n, can be modeled using well-founded recursion. The probabilistic algorithms that do not satisfy the above conditions but are sure to terminate can be modeled by the *probabilistic while loop* proposed in [15].

The probabilistic programs can also be expressed in the more general state-transforming monad where the states are the infinite Boolean sequences.

```
⊢ ∀ a,s.  unit a s = (a,s)
⊢ ∀ f,g,s.  bind f g s = let (x,s')← f(s) in g x s'
```

The `unit` operator is used to lift values to the monad, and the `bind` is the monadic analogue of function application. All the monad laws hold for this definition, and the notation allows us to write functions without explicitly mentioning the sequence that is passed around, e.g., function *bit* can be defined as

```
⊢ bit_monad =
        bind sdest (λb.  if b then unit 1 else unit 0)
```

where `sdest` gives the head and tail of a sequence as a pair (*shd* `s`, *stl* `s`).

Hurd [15] also formalized some mathematical measure theory in HOL in order to define a probability function $\mathbb{P}$ from sets of infinite Boolean sequences to real numbers between 0 and 1. The domain of $\mathbb{P}$ is the set $\mathcal{E}$ of events of the probability. Both $\mathbb{P}$ and $\mathcal{E}$ are defined using the Carathéodory's Extension theorem, which ensures that $\mathcal{E}$ is a σ-algebra: closed under complements and countable unions. The formalized $\mathbb{P}$ and $\mathcal{E}$ can be used to derive the basic laws of probability in the HOL prover, e,g., the additive law, which represents the probability of two disjoint events as the sum of their probabilities:

```
⊢ ∀ A B. A ∈ ℰ ∧ B ∈ ℰ ∧ A ∩ B = ∅ ⇒
                              ℙ(A ∪ B) = ℙ(A) + ℙ(B)
```

The formalized $\mathbb{P}$ and $\mathcal{E}$ can also be used to prove probabilistic properties for probabilistic programs such as

$$\vdash \mathbb{P}\ \{\mathtt{s}\ |\ \mathtt{fst\ (bit\ s)\ =\ 1}\} = \tfrac{1}{2}$$

where the function `fst` selects the first component of a pair.

The measurability of a function is an important concept in probability theory and also a useful practical tool for proving that sets are measurable [3]. In Hurd's formalization of probability theory, a set of infinite Boolean sequences, S, is said to be measurable if and only if it is in $\mathcal{E}$, i.e., $\mathrm{S} \in \mathcal{E}$. Since the probability measure $\mathbb{P}$ is only defined on sets in $\mathcal{E}$, it is very important to prove that sets that arise in verification are measurable. Hurd [15] showed that a function is guaranteed to be measurable if it accesses the infinite boolean sequence using only the `unit`, `bind` and `sdest` primitives and thus leads to only measurable sets.

Hurd formalized a few discrete random variables and proved their correctness by proving the corresponding PMF properties [15]. Because of their discrete nature, all these random variables either compute a finite number of values or are sure to terminate. Thus, they can be expressed using Hurd's methodology by either well formed recursive functions or the probabilistic while loop [15]. On the other hand, continuous random variables always compute an infinite number of values and therefore would require all the random bits in the infinite Boolean sequence if they are to be represented using Hurd's methodology. The corresponding deterministic functions cannot be expressed by either recursive functions or the probabilistic while loop and it is mainly for this reason that the specification of continuous random variables needs to be handled differently than their discrete counterparts.

3 Formalization of the Standard Uniform Distribution

In this section, we present the formalization of the Standard Uniform distribution that is the first step in the proposed methodology for the formalization of continuous probability distributions as shown in Figure 1. The Standard Uniform random variable is a continuous random variable for which the probability that it will belong to a subinterval of [0,1] is proportional to the length of that subinterval. It can be characterized by the CDF as follows:

$$Pr(X \leq x) = \begin{cases} 0 & \text{if } x < 0; \\ x & \text{if } 0 \leq x < 1; \\ 1 & \text{if } 1 \leq x. \end{cases} \tag{2}$$

3.1 Formal Specification of Standard Uniform random variable

The Standard Uniform random variable can be formally expressed in terms of an infinite sequence of random bits as follows [13]

$$\lim_{n \to \infty} (\lambda n. \sum_{k=0}^{n-1} (\frac{1}{2})^{k+1} X_k) \tag{3}$$

where, X_k denotes the outcome of the k^{th} random bit; $true$ or $false$ represented as 1 or 0 respectively. The mathematical relation of Equation (3) can be formalized in the HOL theorem prover in two steps. The first step is to formalize a discrete Standard Uniform random variable that produces any one of the equally spaced 2^n dyadic rationals in the interval $[0, 1 - (\frac{1}{2})^n]$ with the same probability $(\frac{1}{2})^n$. This random variable can be formalized by a recursive function using Hurd's methodology as it consumes a finite number of random bits, i.e., n.

```
⊢ (std_unif_disc 0 = unit 0) ∧
   ∀ n.  (std_unif_disc (suc n) =
            bind (std_unif_disc n) (λm.  bind sdest
                 (λb.  unit (if b then ((1/2)^(n+1) + m) else m))))
```

The function *std_unif_disc* allows us to formalize the real sequence of Equation (3) in the HOL theorem prover. Now, the formalization of the mathematical concept of limit of a real sequence in HOL [12] can be used to formally specify the Standard Uniform random variable of Equation (3) as follows

```
⊢ ∀ s.  std_unif_cont s = lim (λn.  fst(std_unif_disc n s))
```

where *lim* is the HOL function for the limit of a real sequence [12].

3.2 Formal Verification of Standard Uniform random variable

The formalized Standard Uniform random variable, *std_unif_cont*, can be verified to be correct by proving its CDF to be equal to the theoretical value given in Equation 2 and its PMF to be equal to 0, which is an intrinsic characteristic of all continuous random varaibles. For this purpose, it is very important to prove that sets, $\{s \mid std\_unif\_cont\ s\ \leq\ x\}$ and $\{s \mid std\_unif\_cont\ s\ =\ x\}$, that arise in this verification are measurable, i.e., they are in $\mathcal{E}$. It has been shown in [15] that if a function accesses the infinite boolean sequence using only the `unit`, `bind` and `sdest` primitives then the function is guaranteed to be measurable and thus leads to measurable sets. The function std_unif_disc satisfies this condition and thus Hurd's formalization framework can be used to prove

```
Lemma 3.1:
⊢ ∀ x,n.  {s | FST (std_unif_disc n s) ≤ x} ∈ ℰ ∧
          {s | FST (std_unif_disc n s) = x} ∈ ℰ
```

On the other hand, the definition of the function std_unif_cont involves the *lim* function and thus the corresponding sets can not be proved to be measurable in a very straight forward manner. Therefore, in order to prove this, we leveraged the fact that each set in the sequence of sets $(\lambda n.\{s \mid FST(std\_unif\_disc\, n\, s) \leq\ x\})$ is a subset of the set before it, in other words, this sequence of sets is a monotonically decreasing sequence. Thus, the countable intersection of all sets in this sequence can be proved to be equal to the set $\{s \mid std\_unif\_cont\ s\ \leq\ x\}$

```
Lemma 3.2:
⊢ ∀ x.  {s | std_unif_cont s ≤ x} =
             ⋂_n (λ n.  {s | FST (std_unif_disc n s) ≤ x})
```

Now the set $\{s \mid std\_unif\_cont\ s\ \leq\ x\}$ can be proved to be measurable since $\mathcal{E}$ is closed under countable intersections [15] and all the sets in the sequence $(\lambda n.\{s \mid FST(std\_unif\_disc\, n\, s)\ \leq$ $x\})$ are measurable according to Lemma 1. Using a similar reasoning, the set $\{s \mid std\_unif\_cont\ s\ =$ $x\}$ can also be proved to be measurable.

```
Theorem 3.1:
⊢ ∀ x.  {s | std_unif_cont s ≤ x} ∈ ℰ ∧
      {s | std_unif_cont s = x} ∈ ℰ
```

It is important to note that, because of the closed under complements and countable unions property of $\mathcal{E}$, Theorem 3.1 can be used to prove any set that involves a relational property of the function *std_unif_cont*, e.g. $\{s \mid \mathit{std\_unif\_cont}\ s\ < x\}$ and $\{s \mid \mathit{std\_unif\_cont}\ s\ \geq\ x\}$ e.t.c, to be measurable.

Theorem 3.1 and some real analysis formalization can be used to verify the correctness of the function *std_unif_cont* in the HOL theorem prover by proving its CDF to be the same as Equation (2) and its PMF to be equal to 0 [13]. The HOL theory corresponding to this verification is given in Appendix A.

```
Theorem 3.2:
⊢ ∀ x.  ℙ{s | std_unif_cont s ≤ x} =
        if (x < 0) then 0 else (if (x < 1) then x else 1)

Theorem 3.3:
⊢ ∀ x.  ℙ{s | std_unif_cont s = x} = 0
```

4 Formalization of the Cumulative Distribution Function

In this section, we present the formal specification of the CDF and the verification of CDF properties in the HOL theorem prover. It is the second step in the proposed methodology for the formalization of continuous probability distributions as shown in Figure 1.

4.1 Formal Specification of CDF

It follows from Equation (1) that the CDF for any random variable, R, is a function, F_R, defined on the real line. Therefore, the CDF can be formally specified in HOL by a higher-order-logic function that accepts a random variable and a real argument and returns the probability of the event when the given random variable is less than or equal to the value of the given real number. Hurd's formalization of the probability function ℙ, which maps sets of infinite Boolean sequences to real numbers between 0 and 1, can be used to formally specify the CDF as follows:

```
⊢ ∀ R x.  CDF R x = ℙ {s | R s ≤ x}
```

4.2 Formal Verification of CDF Properties

In this section, we present the formal verification of the CDF properties [17] within the HOL theorem prover. These formalized properties not only ensure the correctness of our CDF specification but also play a vital role in proving the correctness of the ITM in Section 5 and determining probabilities associated with various events while analyzing probabilistic systems. All the following properties are verified using the HOL set, measure and probability theories [15] along with the HOL formalization of real analysis [12] and under the assumption that the sets $\{s \mid R\ s \leq x\}$ and $\{s \mid R\ s = x\}$ are measurable, that is, they belong to the set $\mathcal{E}$. The HOL theory corresponding to this verification is given in Appendix A.

4.2.1 CDF Bounds

For any real number x, $0 \leq F_R(x) \leq 1$.

```
Theorem 4.1:
⊢ ∀ R x.  (0 ≤ CDF R x) ∧ (CDF R x ≤ 1)
```

4.2.2 CDF is Monotonically Increasing

For any two real numbers a and b, if $a < b$*, then* $F_R(a) \leq F_R(b)$.

```
Theorem 4.2:
⊢ ∀ R a b.  a < b ⇒ (CDF R a ≤ CDF R b)
```

4.2.3 Interval Probability

For any two real numbers a and b, if $a < b$ *then* $Pr(a < R \leq b) = F_R(b) - F_R(a)$

```
Theorem 4.3:
⊢ ∀ R a b.  a < b ⇒
                     (ℙ {s | (a < R s) ∧ (R s ≤ b) } = CDF R b - CDF R a)
```

4.2.4 CDF at Positive Infinity

$\lim_{x \to \infty} F_R(x) = 1;$ *that is,* $F_R(\infty) = 1$

```
Theorem 4.4:
⊢ ∀ R. lim (λ n.  CDF R (&n)) = 1
```

where, *lim M* represents the HOL formalization of the limit of a real sequence [12], such that *lim M* is the limit value of the real sequence M (i.e., $\lim_{n \to \infty} M(n) = lim\ M$).

4.2.5 CDF at Negative Infinity

$\lim_{x \to -\infty} F_R(x) = 0;$ *that is,* $F_R(-\infty) = 0$

```
Theorem 4.5:
⊢ ∀ R. lim (λ n.  CDF R (-&n)) = 0
```

4.2.6 CDF is Continuous from the Right

For every real number a, $\lim_{x \to a^+} F_R(x) = F_R(a)$*, where* $\lim_{x \to a^+} F_R(x)$ *is defined as the limit of* $F_R(x)$ *as x tends to a through values greater than a. Since* F_R *is monotone and bounded, this limit always exists.*

Theorem 4.6:
⊢ ∀ R a. lim (λ n. CDF R (a + $\frac{1}{\&(n+1)}$)) = CDF R a

4.2.7 CDF Limit from the Left

For every real number a, $\lim_{x \to a^-} F_R(x) = Pr(R < a)$*, where* $\lim_{x \to a^-} F_R(x)$ *is defined as the limit of* $F_R(x)$ *as* x *tends to* a *through values less than* a*.*

Theorem 4.7:
$\vdash$ $\forall$ R a. lim (λ n. CDF R (a - $\frac{1}{\&(n+1)}$)) = $\mathbb{P}$ {s | (R s < a}}

4.3 Determining Interval Probabilities

The CDF of a random variable, R, can be used to determine the probability that R will lie in any specified interval of the real line. In this section, we show how this can be done in the HOL theorem prover by splitting the real line in three disjoint intervals; $(-\infty, a]$, $(a, b]$, (b, ∞). We also consider the special case of using CDF to determine the PMF of a given random variable.

The CDF with a real argument a can be used directly to find the probability that the corresponding random variable lies in the interval $(-\infty, a]$. Whereas, the probability that a random variable lies in the interval $(a, b]$ can be determined by its CDF values for the real arguments a and b as has been proved in Theorem 4.3. The probability of a random variable lying in the third interval can also be expressed in terms of the CDF by using the set and probability theories

Theorem 4.8:
$\vdash$ $\forall$ R b. $\mathbb{P}$ {s | b < R s} = 1 - (CDF R b)

The PMF of a random variable can also be expressed in terms of the CDF function by using the fact that for any real value a the set of infinite Boolean sequences $\{s \mid R\ s \leq a\}$ is equal to the union of the sets $\{s \mid R\ s < a\}$ and $\{s \mid R\ s = a\}$. Now, using the additive law of the probability function $\mathbb{P}$, given in Section 2.2, and Theorems 4.6 and 4.7, we were able to prove

Theorem 4.9:
$\vdash$ $\forall$ R a. $\mathbb{P}$ {s | R s = a} =
lim (λ n. CDF R (a + $\frac{1}{\&(n+1)}$)) - lim (λ n. CDF R (a - $\frac{1}{\&(n+1)}$))

A unique characteristic for all continuous random variables is that their PMF is equal to 0. Theorem 4.9 along with the formalization of continuous functions [12] allowed us prove this property in the HOL theorem prover.

Theorem 4.10:
$\vdash$ $\forall$ R a. ($\forall$x. (λx. CDF R x) contl x) $\Rightarrow$
($\mathbb{P}$ {s | R s = a} = 0)

where, ($\forall$ $x.f$ *contl* x) represents the HOL function definition for a continuous function [12] such that the function f is continuous for all x.

5 Formalization of the Inverse Transform Method

In this section, we present the formal specification of the inverse function for a CDF and the verification of the ITM in the HOL theorem prover. It is the third step in the proposed methodology for the formalization of continuous probability distributions as shown in Figure 1.

The ITM is based on the following proposition [7].

> *Let U be a Standard Uniform random variable. For any continuous CDF F, the random variable X defined by $X = F^{-1}(U)$ has CDF F, where $F^{-1}(U)$ is defined to be the value of x such that $F(x) = U$.*

Mathematically,

$$Pr(F^{-1}(U) \leq x) = F(x) \tag{4}$$

5.1 Formal Specification of the Inverse Transform method

We formalized the mathematical concept of inverse function for a CDF in HOL as a predicate *inv_cdf_fn* which accepts two functions, f and g, of type ($real \rightarrow real$) and returns true if and only if the function f is the inverse of the CDF g according to the above proposition.

```
⊢ ∀ f g.  inv_cdf_fn f g =
      (∀x.  (0 < g x ∧ g x < 1) ⇒ (f (g x) = x) ∧
                    (∀x.  0 < x ∧ x < 1 ⇒ (g (f x) = x))) ∧
      (∀x.  (g x = 0) ⇒ (x ≤ f (0))) ∧
      (∀x.  (g x = 1) ⇒ (f (1) ≤ x))
```

The predicate *inv_cdf_fn* considers three separate cases, the first one corresponds to the strictly monotonic region of the CDF, i.e., when the value of the CDF is between 0 and 1 and the next two correspond to the flat regions of the CDF, i.e, when the value of the CDF is either equal to 0 or 1 respectively. These three cases cover all the possible values of a CDF as according to Theorem 4.1 the value of CDF can never be less than 0 or greater than 1.

The inverse of a function f, $f^{-1}(u)$, is defined to be the value of x such that $f(x) = u$. More formally, if f is a one-to-one function with domain X and range Y, its inverse function f^{-1} has domain Y and range X and is defined by $f^{-1}(y) = x \Leftrightarrow f(x) = y$, for any y in Y. The composition of inverse functions yields some very interesting results.

$$f^{-1}(f(x)) = x \ for\ all\ x \in X, \quad f(f^{-1}(x)) = x \ for\ all\ x \in Y \tag{5}$$

We used the above characteristic of inverse functions in the predicate *inv_cdf_fn* for the strictly monotonic region of the CDF as the CDF in this region is a one-to-one function.

One the other hand, in the flat regions of the CDF, i.e., when the CDF is either 0 or 1, the CDF is not injective. Consider the example of some CDF, F, which returns 0 for a real argument a. From Theorems 4.1 and 4.2, we know that the CDF F will also return 0 for all real arguments that are less than a as well, i.e., $\forall x.x \leq a \Rightarrow F(x) = 0$. Therefore, no inverse function satisfies the conditions of Equation (5) for the CDF in the flat regions. This issue is usually resolved in the texts of nonuniform random number generation methods by defining the inverse function of a CDF in such a way that it returns the infimum (inf) or the supremum (sup) of all the possible values of the real argument for which the CDF is equal to a given value, i.e., $f^{-1}(u) = inf\{x|f(x) = u\}$ or $f^{-1}(u) = sup\{x|f(x) = u\}$ [7], where f represents the CDF. Even though this approach has been shown to analytically verify the correctness of the ITM in many text books [7], it was not found to be sufficient enough for a formal definition in the HOL theorem prover. If inf function is used to define the inverse function then the problem arises for the case when the value of the CDF is equal to 0. For this case, the set $\{x|f(x) = 0\}$ becomes unbounded at the lower end because of the CDF property given in Theorem 4.5 and thus the value of the inverse function becomes undefined. Similarly, if the *sup* function is used to define the inverse function, the value of the inverse function becomes undefined for the case when the value of the CDF is equal to 1. In order to overcome this problem, we defined the inverse function of a CDF in the predicate *inv_cdf_fn* separately for the two flat regions, i.e., it returns the maximum value of all the arguments for which the CDF is equal to 0 and the minimum value of all the arguments for which the CDF is equal to 1.

5.2 Formal Verification of the Inverse Transform method

The correctness theorem for the ITM can be expressed in the HOL theorem prover as follows:

```
Theorem 5.1:
⊢ ∀ f g x.  (is_cont_cdf_fn g) ∧ (inv_cdf_fn f g) ⇒
                                (ℙ {s | f (std_unif_cont s) ≤ x} = g x)
```

The antecedent of the above implication checks if the function f is a valid inverse function of a continuous CDF g. The predicate *inv_cdf_fn* has been described in the last section and it ensures that the function f is a valid inverse of the CDF g. The predicate *is_cont_cdf_fn* accepts a real valued function, g, of type ($real \rightarrow real$) and returns true if and only if it represents a continuous CDF. A real-valued function can be characterized as a continuous CDF if it is a continuous function and satisfies the CDF properties given in Theorems 4.2, 4.4 and 4.5. Therefore, the predicate *is_cont_cdf_fn* is defined in the HOL theorem prover as follows:

```
⊢ ∀ g.  is_cont_cdf_fn g = (lim (λ n.  g (&n)) = 1) ∧
                            (lim (λ n.  g (-&n)) = 0) ∧
                            (∀ a b.  a < b ⇒ g a ≤ g b) ∧
                            (∀ x.  (λx.  g x) contl x)
```

Where *contl* represents the HOL function definition for a continuous function formalized in [12].

The conclusion of the implication in Theorem 5.1 represents the correctness proof of the ITM given in Equation (4). The function *std_unif_cont* in this theorem is the formal definition of the Standard Uniform random variable, described in Section 3. Theorem 3.2 can be used to simplify the proof goal of Theorem 5.1 to the following subgoal:

```
Lemma 5.1:
⊢ ∀ f g x.  (is_cont_cdf_fn g) ∧ (inv_cdf_fn f g) ⇒
    (ℙ {s | f (std_unif_cont s) ≤ x} = ℙ {s | std_unif_cont s ≤ g x})
```

Next, we use the theorems of Section 3 and 4 along with the formalized measure and probability theories [15] to prove that the sets that arise in this verification are measurable, i.e., they are in $\mathcal{E}$.

```
Lemma 5.2:
⊢ ∀ f g x.  (is_cont_cdf_fn g) ∧ (inv_cdf_fn f g) ⇒
                                  ({s | f (std_unif_cont s) ≤ x} ∈ ℰ) ∧
                                  ({s | std_unif_cont ) ≤ g x} ∈ ℰ) ∧
                                  ({s | f (std_unif_cont s) = x} ∈ ℰ)
```

The subgoal of Lemma 5.1 can now be proved using Lemma 5.2, the theorems from Section 3 and 4 and the formalization of probability theory [15]. The HOL theory corresponding to this verification is given in Appendix A. The main advantage of the formally verified ITM (i.e. Theorem 5.1) is that the complex proof goal of verifying the CDF property of a random variable, which involves reasoning based on the measure and probability theories, formalization of the Standard Uniform random variable and some real analysis, can be broken down in two simpler sub goals which only involve reasoning based on real analysis; i.e, (1) Verifying that a function g, of type ($real \rightarrow real$), represents a valid CDF and (2) Verifying that another function f, of type ($real \rightarrow real$), is a valid inverse of the CDF g.

6 Formalization of Continuous Probability Distributions

In this section, we present the formal specification of four continuous random variables; Uniform, Exponential, Rayleigh and Triangular and verify the correctness of these random variables by proving their corresponding CDF properties in the HOL theorem prover.

6.1 Formal Specification of Continuous Random Variables

All continuous random variables, for which the inverse of the CDF exists in a closed mathematical form, can be expressed in terms of the Standard Uniform random variable according to the ITM proposition given in Section 5 [7]. We selected four such commonly used random variables which are formally expressed in terms of the formalized Standard Uniform random variable (*std_unif_cont*) in Table 2. The functions *ln*, *exp*, *sqrt* and *pow* in the formalized definitions are the HOL functions for *logarithm*, *exponential*, *square root* and *power* respectively [12] and the symbols l and *sig* in the formalized definitions have been used for the constants λ and σ.

6.2 Formal Verification of Continuous Random Variables

In this section, we illustrate the process of using the correctness theorem of the ITM, formalized in Section 5, to verify the CDF and measurability properties of a continuous random variable for which the inverse of the CDF exists in a closed mathematical form. The first step in this regard is to express the given continuous random variable as $F^{-1}(U\ s)$, where, F^{-1} is a function of type ($real \rightarrow real$) and U represents the formalized Standard Uniform

Distribution	**CDF**	**Formalized Random Variable**
Exponential(λ)	0 if $x \leq 0$; $1 - exp^{-\lambda x}$ if $0 < x$.	$\vdash \forall s\ l.\ exp\_rv\ l\ s\ =$ $-\frac{1}{l} * ln(1 - std\_unif\_cont\ s)$
Uniform(a, b)	0 if $x \leq a$; $\frac{x-a}{b-a}$ if $a < x \leq b$; 1 if $b < x$.	$\vdash \forall s\ l.\ uniform\_rv\ a\ b\ s\ =$ $(b - a) * (std\_unif\_cont\ s) + a$
Rayleigh(σ)	0 if $x \leq 0$; $1 - exp^{\frac{-x^2}{2\sigma^2}}$ if $0 < x$.	$\vdash \forall s\ l.\ rayleigh\_rv\ sig\ s\ =$ $sig * sqrt(-2 * ln(1 - std\_unif\_cont\ s))$
Triangular($0, a$)	0 if $x \leq 0$; $(\frac{2}{a}(x - \frac{x^2}{2a}))$ if $x < a$; 1 if $a \leq x$.	$\vdash \forall s\ a\ .\ triangular\_rv\ l\ s\ =$ $a * (1 - sqrt(1 - std\_unif\_cont\ s))$

Table 2: Continuous Random Variables (CDF^{-1} exists)

random variable. For example, the Exponential random variable given in Table 2 can be expressed as $((\lambda x. -\frac{1}{l} * ln(1 - x))(std\_unif\_cont\ s))$. Similarly, we can express the CDF of the given random variable as $F(x)$, where F is a function of type $(real \rightarrow real)$ and x is a real data type value. For example, the CDF of the Exponential random variable given in Table 2 can be expressed as $((\lambda x.(if\ x \leq 0\ then\ 0\ else\ 1 - exp^{-\lambda x}))\ x)$.

The next step is to prove that the function F represents a valid continuous CDF and the function F^{-1} is a valid inverse function of the CDF F. The predicates *is_cont_cdf_fn* and *inv_cdf_fn*, defined in Section 5, can be used for this verification and the corresponding theorems for the Exponential random variable are given below

Lemma 6.1:
$\vdash$ is_cont_cdf_fn (λx. if x $\leq$ 0 then 0 else (1 - exp (-l * x)))

Lemma 6.2:
$\vdash$ inv_cdf_fn (λ x. -$\frac{1}{l}$ * ln (1 - x))
(λx. if x $\leq$ 0 then 0 else (1 - exp (-l * x)))

Now, Theorem 5.1 and Lemma 5.2 can be used to verify the CDF and the measurability of the sets corresponding to the given continuous random variable respectively. These theorems for the Exponential random variable are given below

Theorem 6.1:
$\vdash$ l x. (0 < l) $\Rightarrow$
($\mathbb{P}$ {s | exp_rv r s $\leq$ x} =
(if x $\leq$ 0 then 0 else (1 - exp (-l * x))))

Theorem 6.2:
$\vdash$ l x. (0 < l) $\Rightarrow$
({s | exp_rv r s $\leq$ x} $\in \mathcal{E}$ $\wedge$ ({s | exp_rv r s = x} $\in \mathcal{E}$

The above results allow us to formally reason about interesting probabilistic properties of continuous random variables within a higher-order-logic theorem prover. The measurability of the sets $\{s|\ F^{-1}(U\ s) \leq x\}$ and $\{s|\ F^{-1}(U\ s) = x\}$ can be used to prove that any set that involves a relational property with the random variable $(F^{-1}(U\ s))$, e.g. $\{s \mid F^{-1}(U\ s) < x\}$ and $\{s \mid F^{-1}(U\ s) \geq x\}$, is measurable because of the closed under complements and countable unions property of $\mathcal{E}$. The CDF properties can then be used to determine probabilistic quantities associated with these sets as has been shown in Section 4.

The CDF and measurability properties of the rest of the continuous random variables given in Table 2 can also be proved in a similar way and the HOL theory corresponding to this verification is given in Appendix A. For illustration purposes, the final theorems which are proved using the real number theories in HOL [12] are given below:

```
Theorem 6.3:
 ⊢ a b x.  (a < b) ⇒
     ℙ {s | uniform_rv a b s ≤ x} =
         if x ≤ a then 0 else (if x < b then (x - a) / (b - a) else 1)

Theorem 6.4:
⊢ x sig.  (0 < sig) ⇒
     ℙ {s | rayleigh_rv sig s ≤ x} =
         (if x ≤ 0 then 0 else (1 - $\frac{exp(x^2)}{(2*sig^2)}$))

Theorem 6.5:
⊢ a x.  (0 < a) ⇒
     ℙ {s | triangular_rv a s ≤ x} =
         (if (x ≤ 0) then 0 else (if (x < a) then
             ($\frac{2}{a}$ * (x - $\frac{x^2}{2*a}$)) else 1))
```

7 Potential Applications

In this section, we present some of the electrical engineering and computer science applications which can be formally expressed and reasoned about using the formalized continuous random variables of Section 6.

A distinguishing characteristic of the proposed probabilistic analysis approach is the ability to perform precise quantitative analysis of probabilistic systems. In this section, we first illustrate this statement by considering a simple probabilistic analysis example. Then, we present some probabilistic systems which can be formally analyzed using the formalized continuous random variables.

Consider the problem of determining the probability of the event when there is no incoming request for 10 seconds in a Web server. Assume that the *interarrival* time of incoming requests is known, from statistical analysis, and is exponentially distributed with an average rate of requests $\lambda = 0.1$ jobs per second. The given problem can be solved in the HOL theorem prover by finding the probability of the event when the value of the Exponential random variable, with parameter 0.1 (i.e., $\lambda = 0.1$), lies in the interval $[10, \infty)$. The probability for this event can be expressed in terms of the CDF of the Exponential random variable by using the measurability property proved in Theorem 6.2 and the set and probability theories in HOL.

```
⊢ ℙ {s | 10 < exp_rv 0.1 s} = 1 - (cdf (λs.  exp_rv 0.1 s) 10)
```

The CDF of the Exponential random variable given in Theorem 6.1 can now be used to simplify the right hand side of the above equation to be equal to $exp(-1)$. Thus, we were able to determine the unknown probability with 100% precision; a novelty which is not available in the simulation based approaches. The higher-order-logic theorem proving based probabilistic analysis can be applied to a variety of different domains and some of these potential application areas have been mentioned below.

The sources of error in computer arithmetic operations are basically quantization operations and are modeled as uniformly distributed continuous random variables [29]. A number of successful attempts have been made to perform the statistical analysis of computer arithmetic analytically or by simulation, e.g., [16]. These kind of analysis form a very useful case study for our formalized continuous Uniform distribution as the formalization of both floating point and fixed point numbers already exists in the HOL theorem prover [1].

Exponential distribution is often used in queuing theory applications because of its memoryless property [28]. We can utilize the formalized Exponential random variable along with a formalized Poisson random variable to formalize the Birth-Death process which is a special kind of Continuous-Time Markov Chain used in modeling queuing systems. The higher-order-logic formalization of the Birth-Death process may open the door for the formalized probabilistic analysis of a wide range of telecommunication and computer network protocols, e.g., the CSMA/CD protocol [8], the IEEE 802.11 wireless LAN protocol [19] e.t.c.

The formalized continuous random variables can also be used to compare the efficiency of various algorithms for NP-complete problems [23] in the HOL theorem prover.

The Rayleigh distribution usually arises when a two dimensional vector has its two orthogonal components normally and independently distributed. The formalized Rayleigh distribution can be used to perform the formalized probabilistic analysis of the commonly encountered scenario of scattered signals reaching a telecommunication receiver by multiple paths.

8 Related Work

Due to the vast application domain of continuous random variables, many researchers around the world are trying to improve the modeling techniques for continuous probability distributions in computer based environments. The ultimate goal is to come up with a probabilistic analysis framework that includes robust and accurate analysis methods, has the ability to perform analysis for large-scale problems and is easy to use. In this section, we provide a brief account of the state-of-the-art and some related work in this field.

A number of *probabilistic languages*, e.g., `Probabilistic cc` [11], λ_o [24] and `IBAL` [26], have been proposed that are capable of modeling random variables. Probabilistic languages treat probability distributions as primitive data types and abstract from their representation schemes. Therefore, they allow programmers to perform probabilistic computations at the level of probability distributions rather than representation schemes. These probabilistic languages are quite expressive and have been shown to express most continuous probability distributions but they have their own limitations. For example, either they require a special treatment such as the lazy list evaluation strategy in `IBAL` and the limiting process in `Probabilistic cc` or they do not support precise reasoning as in the case of λ_o. The proposed theorem proving based approach, on the other hand, is not only capable of formally expressing most continuous probability distributions but also to precisely reason about them.

It is interesting to note that the probabilistic language, λ_o, [24] is based on sampling functions. A sampling function is defined as a mapping from the unit interval [0,1] to a probability domain $\mathfrak{D}$. Given a random number drawn from a Standard Uniform distribution, it returns a sample in $\mathfrak{D}$, and thus specifies a unique probability distribution. Thus, this approach is very similar to what we have proposed in this paper, as it also utilizes the Standard Uniform random variable to obtain other continuous random variables. [24] contains sampling algorithms for various continuous random variables which can be utilized to formalize the respective random variables in the HOL theorem prover using our formalized Standard Uniform random variable.

Another alternative for formal probabilistic verification is to use probabilistic model checking techniques, e.g., [2], [27]. Like the traditional model checking, it involves the construction of a precise mathematical model of the probabilistic system which is then subjected to exhaustive analysis to verify if it satisfies a set of formal properties. This approach is capable of providing precise solutions in an automated way; however it is limited for systems that can only be expressed as a probabilistic finite state machine. Our proposed theorem proving based approach, in contrast, is capable of handling all kinds of probabilistic systems including the *unbounded* ones, as demonstrated by the example in Section 7. Another major limitation of the probabilistic model checking approach is the state space explosion [6], which is not an issue with our approach.

9 Conclusions

In this report, we have proposed to use higher-order-logic theorem proving for probabilistic analysis as an alternative to the state-of-the-art simulation based techniques. We believe that because of the formal nature of the models the analysis will be free of approximation errors, which makes the proposed approach very useful for the performance and reliability optimization of safety critical and highly sensitive engineering and scientific applications.

We presented a methodology for the formalization of continuous probability distributions which is a significant step towards the development of a formal probabilistic analysis framework. Based on this methodology, we described the construction details of a framework for the formalization of all continuous probability distributions for which the inverse of the CDF can be expressed in a closed mathematical form. We demonstrated the practical effectiveness of our framework by formalizing four continuous probability distributions; Uniform, Exponential, Rayleigh and Triangular. To the best of our knowledge, this is the first time that a successful attempt has been made to formalize continuous probability distributions in a higher-order-logic theorem prover.

For our verification, we utilized the HOL theories of *Boolean Algebra*, *Sets*, *Natural Numbers*, *Real Numbers*, *Measure* and *Probability*. Our results can therefore be used as an evidence for the soundness of the existing HOL libraries and usefulness of theorem provers in proving pure mathematical concepts. The presented formalization can be utilized for the formalization of a number of other mathematical concepts as well. For example, the formalized CDF properties can be used along with the formalization of the mathematical concept of a derivative [12] to formalize the Probability Density Function, which is a very significant characteristic of continuous random variables and can be used to formalize the corresponding statistical quantities. Similarly, the formalization of the Standard Uniform random variable can also be transformed to formalize other continuous probability distributions, for which the inverse CDF is not available in a closed mathematical form, by exploring the formalization of other nonuniform random number generation techniques such as Box-Muller and acceptance/rejection [7].

References

[1] B. Akbarpour, A. Dekdouk, and S. Tahar. Formalization of Cadence SPW Fixed-Point Arithmetic in HOL. In *IFM*, pages 185–204, 2002.

[2] C. Baier, B. Haverkort, H. Hermanns, and J. P. Katoen. Model Checking Algorithms for Continuous time Markov chains. *IEEE Transactions on Software Engineering*, 29(4):524–541, 2003.

[3] P. Billingsley. *Probability and Measure.* John Wiley.

[4] P. Bratley, B. L. Fox, and L. E. Schrage. *A Guide to Simulation.* Springer-Verlag, 1987.

[5] A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[6] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* The MIT Press, 2000.

[7] L. Devroye. *Non-Uniform Random Variate Generation.* Springer-Verlag, 1986.

[8] T . A. Gonsalves and F. A. Tobagi. On the Performance Effects of Station Locations and Access Protocol Parameters in Ethernet Networks. *IEEE Trans. on Communications*, 36(4):441–449, April 1988.

[9] M. J. C. Gordon. Mechanizing Programming Logics in Higher-0rder Logic. In *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 387–439. Springer-Verlag, 1989.

[10] M. J. C. Gordon and T.F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic.* Cambridge University Press, 1993.

[11] V. T. Gupta, R. Jagadeesan, and P. Panangaden. Stochastic Processes as Concurrent Constraint Programs. In *Principles of Programming Languages*, pages 189–202. ACM Press, 1999.

[12] J. Harrison. *Theorem Proving with the Real Numbers.* Springer-Verlag, 1998.

[13] O. Hasan and S .Tahar. Formalization of Standard Uniform Random Variable. Technical Report, Concordia University, Montreal, Canada, December, 2006.

[14] O. Hasan and S .Tahar. Formalization of Continuous Probability Distributions. Technical Report, Concordia University, Montreal, Canada, February, 2007.

[15] J. Hurd. *Formal Verification of Probabilistic Algorithms.* PhD thesis, University of Cambridge, Cambridge, UK, 2002.

[16] T. Kaneko and B. Liu. On Local Roundoff Errors in Floating-Point Arithmetic. *ACM*, 20(3):391–398, 1973.

[17] R. Khazanie. *Basic Probability Theory and Applications.* Goodyear, 1976.

[18] D. E. Knuth. *The Art of Computer Programming*, volume 2. Addison-Wesley Professional, 1998.

[19] A. Köpsel, J. Ebert, and A. Wolisz. A Performance Comparison of Point and Distributed Coordination Function of an IEEE 802.11 WLAN in the Presence of Real-Time Requirements, 2000. Proc. MoMuC.

[20] H. Kuki and W. J. Cody. A Statistical Study of the Accuracy of Floating Point Number Systems. *ACM*, 16(4), 1973.

[21] D. J. C. MacKay. Introduction to Monte Carlo methods. In *Learning in Graphical Models, NATO Science Series*, pages 175–204. Kluwer Academic Press, 1998.

[22] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[23] Panel on Probability and Algorithms, National Research Council. *Probability and Algorithms: Introduction.* National Academy Press, 1992.

[24] S. Park, F. Pfenning, and S. Thrun. A Probabilistic Language based upon Sampling Functions. In *Principles of Programming Languages*, pages 171–182. ACM Press, 2005.

[25] L. C. Paulson. *ML for the Working Programmer.* Cambridge University Press.

[26] A. Pfeffer. IBAL: A Probabilistic Rational Programming Language. In *International Joint Conferences on Artificial Intelligence*, pages 733–740. Morgan Kaufmann Publishers, 2001.

[27] J. Rutten, M. Kwaiatkowska, G. Normal, and D. Parker. Mathematical Techniques for Analyzing Concurrent and Probabilisitc Systems. *CRM Monograph*, 23, 2004.

[28] K. S. Tridevi. *Probability and Statistics with Reliability, Queuing and Computer Science Applications.* Wiley-Interscience, 2002.

[29] B. Widrow. Statistical Analysis of Amplitude-quatized Samled Data Systems. *AIEE Trans. (Applications and Industry)*, 81:555–568, January 1961.

10 Appendix A

In this appendix, we present the HOL implementation of the methodology, illustrated in Figure 1, for the formalization of continuous probability distributions.

We implemented the four major steps illustrated in Figure 1, i.e., the formalization of the Standard Uniform random variable, the Cumulative distribution function (CDF), the Inverse Transform Method (ITM) and Continuous random variables in four different HOL theories. Figure 2 presents the logical dependency of these HOL theories among themselves and to the main existing HOL-4 theories (represented as rectangles) on which they depend.

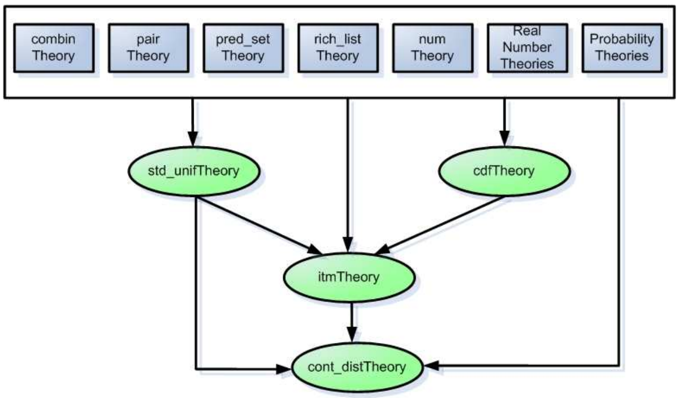

Figure 2: Logical Dependency Graph of the Continuous Probability Distribution Theories

10.1 std_unifTheory

This theory contains the formal definition of the Standard Uniform random variable, *std_unif_cont*, along with the formal proofs of its CDF, Probability Mass Function (PMF) and measurability properties.

10.1.1 Signature

Functions	**Type**
ceiling	*real* → *num*
std_unif_disc	*num* → (*num* → *bool*) → (*real* x (*num* → *bool*))
unif_two_pow	*num* → (*num* → *bool*) → (*num* x (*num* → *bool*))
std_unif_cont	(*num* → *bool*) → *real*
all_std_unif_disc_le	*real* → ((*num* → *bool*) → *bool*)
all_std_unif_disc_eq	*real* → ((*num* → *bool*) → *bool*)

10.1.2 Definitions

ceiling_def

$\vdash_{def}$ `∀ x n. ceiling x = LEAST n. x ≤ &n`

std_unif_disc_def

```
⊢def   ∀ s. (std_unif_disc (0:num) s = (0,s)) ∧
            ∀ s n. (std_unif_disc (SUC n) s =
                ((if shd (SND (std_unif_disc n s)) then
                ((1/2) pow (SUC n) + FST (std_unif_disc n s)) else
                                        FST (std_unif_disc n s),
                stl (SND (std_unif_disc n s)))))
```

unif_two_pow_def

```
⊢def   ∀ s. (unif_two_pow (0:num) s = (0,s)) ∧
            ∀ s n. (unif_two_pow (SUC n) s =
                ((if shd (SND (unif_two_pow n s)) then
                (2 * FST (unif_two_pow n s) + 1) else
                                    2 *  FST (unif_two_pow n s ),
                stl (SND (unif_two_pow n s)))))
```

std_unif_cont_def

```
⊢def  ∀ s. std_unif_cont s =
                lim (λn. FST (std_unif_disc n s))
```

all_std_unif_disc_le_def

```
⊢def  ∀ x. all_std_unif_disc_le x =
            IMAGE (λn. {s | FST (std_unif_disc n s) ≤ x}) UNIV
```

all_std_unif_disc_eq_def

```
⊢def  ∀ x. all_std_unif_disc_eq x =
                IMAGE (λn. {s | (FST (std_unif_disc n s) =
                    & (ceiling (2 pow n * x)) * 1/2 pow n) ∨
                    (FST (std_unif_disc n s) = (& (ceiling
                        (2 pow n * x)) − 1) * 1/2 pow n)}) UNIV
```

10.1.3 Theorems

LAMBDA_LET

```
∀ p f. (λ(m,x). f m x) p = (let (a,b) = p in (f a b))
```

LAMBDA_PAIR

```
∀ m x p f. (λ(m,x). f m x) p = f (FST p) (SND p)
```

REAL_SUB_ASSOC

```
∀ (a: real) b c. a − b − c = a − (b + c)
```

REAL_SUB_ASSOC2

```
∀ (a: real) b c. a − b + c = a + (c − b)
```

LT_SUC_LTE

```
∀ m n. m < (SUC n) = (m < n) ∨ (m = n)
```

HALF_POW_SUC_LE_HALF_POW_N

```
∀ n. (1 / 2) pow SUC n ≤ (1 / 2) pow n
```

HALF_POW_SUC_PLUS_SUCSUC_LE_HALF_POW

```
∀ n. (1 / 2) pow SUC n + (1 / 2) pow SUC (SUC n) ≤ (1 / 2) pow n
```

REAL_NE_LT_GT

```
∀ (a:real) (b:real). ¬(a = b) = (a < b) ∨ (b < a)
```

SUM_HALF_POW_SUC

```
∀ n. sum (0,n) (λn. (1/2) pow SUC n) = 1 − (1/2) pow n
```

REAL_SEQ_LE_EXISTS_EQ

```
∀ (a:num −> real) (b:real) (c:num −> real) (n:num).
            ∃x. (a x = c x) ∧ (a n ≤ b) ∧ (b ≤ c n) ⟹  (b = a x)
```

SIMP_REAL_ARCH

```
∀ x. ∃n. x ≤ &n‘
```

NUM_LT_IMP_ABS_GT_PLUS1_GT

```
∀ (x:real) (n:num).
            (∀m. m < n ⟹  (&m < (abs x))) = (&n < (abs x) + 1 )
```

NUM_LT_IMP_ABS_NLE_PLUS1_GT

```
∀ (x:real) (n:num).
            (∀m. m < n ⟹  ¬(abs x ≤ & m)) = (&n < (abs x) + 1 )
```

HALF_POW_TWO_POW

```
∀ n. (1 / 2) pow n * 2 pow n = 1
```

HALF_POW_TENDSTO_ZERO

```
(λn. (1/2) pow n) −−> 0
```

TWO_HALF_POW_TENDSTO_ZERO

```
(λn. 2 * (1 / 2) pow n) −−> 0
```

REAL_SUB_2

```
∀ m n. & m − & n = (if n ≤ m then & (m − n) else ¬& (n − m))
```

LB_CEILING

```
∀ x. x ≤ &(ceiling x)
```

ABS_PLUS1_GT_CEILING_ABS

```
∀ x. &(ceiling(abs x)) < (abs x) + 1
```

UB_CEILING

```
∀ x. (0 ≤ x) ⟹  &(ceiling(x)) < x + 1
```

CEILING_NUM

```
∀ n. ceiling (&n) = n
```

CEILING_ABS_POS

```
∀ x. 0 ≤ &(ceiling (abs x))
```

CEIL_MONO

```
∀ m n. (0 ≤ m) ∧ (0 ≤ n) ∧ (m ≤ n) ⟹  (ceiling m) ≤ (ceiling n)
```

TWO_POWNX_LE_CEILING_2POWNX

```
∀ (x:real) (n:num).
            ((λn. ((2 pow n) * x) * ((1/2) pow n)) n ≤
             (λn. &(ceiling ((2 pow n) * x)) * ((1/2) pow n)) n)
```

TWO_POWNX_LE_CEILING_2POWNX

```
∀ (x:real) (n:num).
            ((λn. ((2 pow n) * x) * ((1/2) pow n)) n ≤
             (λn. &(ceiling ((2 pow n) * x)) * ((1/2) pow n)) n)
```

CEILING_2POWNX_LE_2POWNX_PLUS1

```
∀ (x:real) (n:num). (0 ≤ x) ⟹
            (λn. (&(ceiling ((2 pow n) * x)) * ((1/2) pow n))) n  ≤
            (λn. (((2 pow n) * x) + 1) * ((1/2) pow n)) n
```

TWO_POW_X_CEIL_HALF_POW_TENDS

```
∀ (x:real). ((λn. ((2 pow n) * x) * ((1/2) pow n)) --> x)
```

TWO_POW_X_PLUS1_HALF_POW_TENDS

```
∀ (x:real). ((λn. ((2 pow n) * x + 1) * ((1/2) pow n)) --> x)
```

UNIQ_NUM_IN_REAL_RANGE_ONE

```
∀ x m n. (x ≤ &n) ∧ (&n < x + 1) ∧
            (x ≤ &m) ∧ (&m < x + 1) ⟹ (&n = &m)
```

CEILING_TWO_POWNX_MONO_SUC_HELPER

```
∀ x m n. (x ≤ &n) ∧ (&n < x + 1) ∧ (x ≤ &m) ⟹  &n ≤ &m
```

CEILING_TWO_POWNX_MONO_SUC

```
∀ n x. (0 ≤ x) ⟹
            (λn. (&(ceiling ((2 pow n) * x)) * 1/2 pow n)) SUC n ≤
            (λn. (&(ceiling ((2 pow n) * x)) * 1/2 pow n)) n
```

CEILING_TWO_POWNX_MONO

```
∀ x. (0 ≤ x) ⟹
            mono (λn. (&(ceiling ((2 pow n) * x)) * ((1/2) pow n)))
```

CEILING_TWO_POWNX_BOUNDED

```
∀ x. (0 ≤ x) ⟹
            bounded(mr1, $≥)
                (λn. (&(ceiling ((2 pow n) * x)) * (1/2 pow n)))
```

CEILING_TWO_POWNX_CONVERGES

```
∀ (x:real). (0 ≤ x) ⟹
                (λn. (&(ceiling ((2 pow n) * x)) *
                                    ((1/2) pow n))) -->
                lim (λn. (&(ceiling ((2 pow n) * x)) *
                                        ((1/2) pow n)))
```

CEILING_TWO_POWNX_CONVERGENT

```
∀ (x:real). (0 ≤ x) ⟹
                (convergent (λn. & (ceiling (2 pow n * x)) *
                                                (1 / 2) pow n))
```

UB_LIM_CEILING_TWO_POWNX

```
∀ (x:real). (0 ≤ x) ⟹
                lim (λn. & (ceiling (2 pow n * x)) *
                                            (1 / 2) pow n) ≤ x
```

LB_LIM_CEILING_TWO_POWNX

```
∀ (x:real) . (0 ≤ x) ⟹
                x ≤ lim (λn. & (ceiling (2 pow n * x)) *
                                                (1 / 2) pow n)
```

LIM_CEILING_TWO_POWNX

```
∀ (x:real). (0 ≤ x) ⟹
                (lim (λn. & (ceiling (2 pow n * x)) *
                                        (1 / 2) pow n) = x)
```

CEILING_TWO_POWNX_TENDSTO_X

```
∀ (x:real). (0 ≤ x) ⟹
                (λn. & (ceiling (2 pow n * x)) *
                                    (1 / 2) pow n) --> x
```

CEIL_TW0_POW_GE_1

```
∀ x n. (0 < x) ⟹  1 ≤ ceiling ((2 pow n) * x)
```

CEILING_TWO_POWNX_MINUS_ONE_TENDSTO_X

```
∀ (x:real). (0 ≤ x) ⟹
               (λn. &(ceiling (2 pow n * x) − 1) *
                                  (1 / 2) pow n) −−> x
```

CEILING_TWO_POWNX_PLUS_ONE_TENDSTO_X

```
∀ (x:real). (0 ≤ x) ⟹
               (λn. &(ceiling (2 pow n * x) + 1) *
                                  (1 / 2) pow n) −−> x
```

CEILING_2POWNX_MINUS1_LT_XHALF_POW

```
∀ n x. (0 ≤ x) ⟹
               (& (ceiling (2 pow n * x)) − 1) * (1/ 2) pow n < x
```

XHALF_POW_LE_CEILING_2POWNX

```
∀ n x. (0 ≤ x) ⟹  x ≤ & (ceiling (2 pow n * x)) * (1 / 2) pow n
```

SND_STD_UNIF_EQ_UNIF_TWO_POW

```
∀ m n s. (SND (std_unif_disc n s)) = (SND (unif_two_pow n s))
```

TWO_POW_STD_UNIF_DISC_EQ_UNIF_TWO_POW

```
∀ s n. (2 pow n) * (FST (std_unif_disc n s)) =
                                  & (FST (unif_two_pow n s))
```

UNIF_TWO_POW_MONAD

```
(unif_two_pow 0 = UNIT (0: num)) ∧
(∀ n. ((unif_two_pow (SUC n)) =
            BIND (unif_two_pow n)
                    (λm. BIND sdest (λb. UNIT
                            (if b then (2 * m + 1) else 2 * m)))))
```

UNIF_TWO_POW_INDEP

```
∀ n. unif_two_pow n IN indep_fn
```

STD_UNIF_DISC_MONAD

```
(std_unif_disc 0 = UNIT (0: real)) ∧
(∀ n. ((std_unif_disc (SUC n)) =
            BIND (std_unif_disc  n)
                    (λm. BIND sdest (λb. UNIT
                    (if b then ((1/2) pow (SUC n) + m) else m)))))
```

STD_UNIF_DISC_INDEP

```
∀ n. std_unif_disc n IN indep_fn
```

UB_STD_UNIF_DISC

```
∀ n s. ((λn. FST (std_unif_disc n s)) n) ≤ 1 − (1/2) pow n
```

STD_UNIF_DISC_LT1

```
∀ n s. ((λn. FST (std_unif_disc n s)) n) < (1: real)
```

LB_STD_UNIF_DISC

```
∀ n s. 0 ≤ ((λn. FST (std_unif_disc n s)) n)
```

STD_UNIF_DISC_BOUNDED

```
∀ s. bounded(mr1, $≥) (λn. FST (std_unif_disc n s))
```

STD_UNIF_DISC_MONO

```
∀ m n. m ≤ n ⟹  (((λn. FST (std_unif_disc n s)) m ≤
                       (λn. FST (std_unif_disc n s)) n))
```

STD_UNIF_DISC_CONVERGENT

```
∀ s. convergent (λn. FST (std_unif_disc n s))
```

STD_UNIF_DISC_SUCN_N_HALF_POW

```
∀ s n. (λn. FST (std_unif_disc n s)) (SUC n) ≤
           (λn. FST (std_unif_disc n s)) n + (1/2) pow (SUC n)
```

STD_UNIF_DISC_M_N_SUM_HALF_POW

```
∀ n s m. n < m ⟹
        (λn. FST (std_unif_disc n s)) m ≤
        (λn. FST (std_unif_disc n s)) n +
           sum (n, m − n) (λn. (1/2) pow (SUC n))
```

STD_UNIF_DISC_DIFFERENCE

```
∀ n s m. n < m ⟹
        (λn. FST (std_unif_disc n s)) m  <
        (λn. FST (std_unif_disc n s)) n + (1 / 2) pow n
```

STD_UNIF_DISC_EQ_EVENTS

```
∀ n x. {s | FST (std_unif_disc n s) = x} IN events bern
```

STD_UNIF_DISC_LE_EVENTS

```
∀ n x. {s | FST (std_unif_disc n s) ≤ x} IN events bern
```

STD_UNIF_DISC_EQ_EVENTS

```
∀ n x. {s | FST (std_unif_disc n s) = x} IN events bern
```

LB_STD_UNIF_CONT

```
∀ s. 0 ≤ std_unif_cont s
```

UB_STD_UNIF_CONT

```
∀ s. std_unif_cont s ≤ 1
```

STD_UNIF_CONT_LE_STD_UNIF_DISC_HALF_POW

```
∀ (s:num −> bool) n.
            std_unif_cont s ≤
                (λn. FST (std_unif_disc n s))n + (1/2) pow n
```

STD_UNIF_CONT_GE_STD_UNIF_DISC

```
∀ (s:num −> bool) n.
            (λn. FST (std_unif_disc n s))n ≤ std_unif_cont s
```

UNIF_DISC_CEIL_SUBSET_CONT

```
∀ x n. (0 ≤ x) ⟹
            {s | FST (std_unif_disc n s) ≤
                (& (ceiling (2 pow n * x)) − 2) * (1/2) pow n}
                    SUBSET {s | std_unif_cont s ≤ x}
```

CONT_SUBSET_UNIF_DISC_CEIL

```
∀ x n. (0 ≤ x) ⟹
            {s | std_unif_cont s ≤ x}
            SUBSET {s | FST (std_unif_disc n s) ≤
                (& (ceiling (2 pow n * x)) * (1/2) pow n)}
```

CONT_SUBSET_UNIF_DISC_LE

```
∀ x n. {s | std_unif_cont s ≤ x}
            SUBSET {s | FST (std_unif_disc n s) ≤ x}
```

CONT_EQX_SUBSET_UNIF_DISC_CEIL_CEIL_SUB1

```
∀ x n. (0 ≤ x)  ⟹
            {s | std_unif_cont s = x} SUBSET
            {s | (FST (std_unif_disc n s) =
                & (ceiling (2 pow n * x)) * (1/2) pow n) ∨
                  (FST (std_unif_disc n s) =
                    (& (ceiling (2 pow n * x)) − 1) *
                                         (1/2) pow n)}
```

IN_ALL_STD_UNIF_DISC_LE

```
∀ x n. {s |  FST (std_unif_disc n s) ≤ x} IN
                                    all_std_unif_disc_le x
```

ALL_STD_UNIF_DISC_LE_ELEMENTS

```
∀ a x. a IN (all_std_unif_disc_le x) ⟹
            ∃n. a = {s |  FST (std_unif_disc n s) ≤ x}
```

ALL_STD_UNIF_DISC_LE_COUNTABLE

```
∀ x. countable (all_std_unif_disc_le x)
```

BIGINTER_ALL_STD_UNIF_DISC_LE_EVENTS_BERN

```
∀ (x:real). BIGINTER (all_std_unif_disc_le x) IN events bern
```

STD_UNIF_CONT_BIGINTER_ALL_STD_UNIF_LE

```
∀ x. {s | std_unif_cont s ≤ x} =
                BIGINTER (all_std_unif_disc_le x)
```

STD_UNIF_CONT_EVENTS_BERN

```
∀ x. {s | std_unif_cont s ≤ x} IN events bern
```

ALL_STD_UNIF_DISC_EQ_ELEMENTS

```
∀ a x. a IN (all_std_unif_disc_eq x) ⟹
            ∃n. a = {s | (FST (std_unif_disc n s) =
                          & (ceiling (2 pow n * x)) * (1/2) pow n) ∨
                          (FST (std_unif_disc n s) =
                          (& (ceiling (2 pow n * x)) − 1) *
                                                   (1/2) pow n)}
```

ALL_STD_UNIF_DISC_EQ_COUNTABLE

```
∀ x. countable (all_std_unif_disc_eq x)
```

BIGINTER_ALL_STD_UNIF_DISC_EQ_EVENTS_BERN

```
∀ (x:real). BIGINTER (all_std_unif_disc_eq x) IN events bern
```

STD_UNIF_CONT_BIGINTER_ALL_STD_UNIF_EQ_GE0

```
∀ x. 0 ≤ x ⟹
        (s | std_unif_cont s = x =
            BIGINTER (all_std_unif_disc_eq x))
```

STD_UNIF_CONT_EQ_EVENTS_BERN_GE0

```
∀ x. 0 ≤ x ⟹
            (s | std_unif_cont s = x IN events bern)
```

STD_UNIF_CONT_EQ_EVENTS_BERN_LT1

```
∀ x. x < 0 ⟹
            (s | std_unif_cont s = x IN events bern)
```

STD_UNIF_CONT_EQ_EVENTS_BERN

```
∀ x. (s | std_unif_cont s = x IN events bern)
```

STD_UNIF_CONT_LT_EVENTS_BERN

```
∀ x. (s | std_unif_cont s < x IN events bern)
```

PROB_UNIF_DISC_CEIL_LE_PROB_CONT

```
∀ x n. (0 ≤ x) ⟹
          (prob bern {s | FST (std_unif_disc n s) ≤
                (& (ceiling (2 pow n * x)) − 2) * (1 / 2) pow n}
          ≤ prob bern {s | std_unif_cont s ≤ x})
```

PROB_CONT_LE_PROB_UNIF_DISC_CEIL

```
∀ x n. (0 ≤ x) ⟹
          prob bern {s | std_unif_cont s ≤ x} ≤
          prob bern {s | FST (std_unif_disc n s) ≤
             (& (ceiling (2 pow n * x)) * (1 / 2) pow n)}
```

PROB_CONT_EQX_LE_PROB_DISC_CEIL_SUB1

```
∀ x n. (0 ≤ x) ⟹
          prob bern {s | std_unif_cont s = x}
          ≤ prob bern {s | (FST (std_unif_disc n s) =
             & (ceiling (2 pow n * x)) * (1 / 2) pow n) ∨
                         (FST (std_unif_disc n s) =
             (& (ceiling (2 pow n * x)) − 1) * (1 / 2) pow n)}
```

CDF_UNIF_DISC_LT0

```
∀ n x. x < 0 ⟹
          (prob bern {s | FST (std_unif_disc n s) ≤ x} = 0)
```

PMF_UNIF_DISC_LT0

```
∀ n x. x < 0 ⟹
          (prob bern {s | FST (std_unif_disc n s) = x} = 0)
```

CDF_UNIF_DISC_GE1

```
∀ n x. 1 ≤ x ⟹
          (prob bern {s | FST (std_unif_disc n s) ≤ x} = 1)
```

PMF_UNIF_DISC_GE1

```
∀ n x. 1 ≤ x ⟹
          (prob bern {s | FST (std_unif_disc n s) = x} = 0)
```

PMF_STD_UNIF_DISC_GE0_LT1

```
∀ n m. (m < (2 ** n)) ⟹
          (prob bern {s | FST (std_unif_disc n s) =
             &m / & (2 ** n)} = 1 / & (2 ** n))
```

CDF_UNIF_DISC_GE0_LT1

```
∀ n m. (m < (2 ** n)) ⟹
          (prob bern {s | FST (std_unif_disc n s) ≤
             &m / & (2 ** n)} = &(SUC m) / & (2 ** n))
```

PROB_UNIF_DISC_CEIL_TWO_POW_BY_TWO_POW

```
∀ x n. (0 ≤ x) ∧ (x < 1) ⟹
            (prob bern {s | FST (std_unif_disc n s) =
                & (ceiling (2 pow n * x)) / 2 pow n}
                                        ≤ (1 / 2) pow n)
```

PROB_DISC_UNIF_EQ_CEIL2POW_OR_CEIL2POW_MINUS1

```
∀ x n. (0 ≤ x) ∧ (x ≤ 1) ⟹
            (prob bern {s | (FST (std_unif_disc n s) =
                & (ceiling (2 pow n * x)) * (1/2) pow n) ∨
                             (FST (std_unif_disc n s) =
               (& (ceiling (2 pow n * x)) − 1) * (1/2) pow n)}
                                            ≤ 2 * (1/2) pow n)
```

PMF_STD_UNIF_CONT_LE_TWICE_HALF_POW

```
∀ x n. (0 ≤ x) ∧ (x ≤ 1) ⟹
        prob bern {s | std_unif_cont s = x} ≤ 2 * (1 / 2) pow n
```

PMF_STD_UNIF_CONT_LE0

```
∀ x. (0 ≤ x) ∧ (x ≤ 1) ⟹
            prob bern {s | std_unif_cont s = x} ≤ 0
```

PROB_LB_UB_STD_UNIF_CONT_RANGE_EQ0

```
∀ x. (0 ≤ x) ∧ (x ≤ 1) ⟹
            (prob bern {s | std_unif_cont s = x} = 0)
```

PMF_STD_UNIF_CONT

```
∀ x. (prob bern {s | std_unif_cont s = x} = 0)
```

PROB_UNIF_DISC_LE_CEIL_TWO_POW_MINUS2

```
∀ x n. (0 ≤ x) ∧ (x < 1) ⟹
            (prob bern {s | FST (std_unif_disc n s) ≤
                (& (ceiling (2 pow n * x)) − 2) * (1 / 2) pow n} =
                    &(ceiling (2 pow n * x) − 1) * (1 / 2) pow n)
```

PROB_STD_UNIF_LE_CEIL_SUC_2POW

```
∀ x n. (0 ≤ x) ∧ (x < 1) ⟹
            (prob bern {s | FST (std_unif_disc n s) ≤
                (& (ceiling (2 pow n * x)) * (1/2) pow n)}
                    ≤ & (ceiling (2 pow n * x) + 1) * (1/2) pow n)
```

PROB_CONT_LE_CEIL_SUC_2POW

```
∀ x n. (0 ≤ x) ∧ (x < 1) ⟹
            prob bern {s | std_unif_cont s ≤ x}
                ≤ & (ceiling (2 pow n * x) + 1) * (1 / 2) pow n
```

CEIL_TWO_POW_MINUS2_LE_PROB_CONT

```
∀ x n. (0 ≤ x) ∧ (x < 1) ⟹
          (&(ceiling (2 pow n * x) − 1) * (1 / 2) pow n
                 ≤ prob bern {s | std_unif_cont s ≤ x})
```

CDF_UNIF_CONT_LT0

```
∀ x. (x < 0)  ⟹
          (prob bern {s | std_unif_cont s ≤ x} = 0)
```

CDF_UNIF_CONT_GE1

```
∀ n x. 1 ≤ x ⟹
          (prob bern {s | std_unif_cont s ≤ x} = 1)
```

CDF_UNIF_CONT_GE0_LT1

```
∀ x. (0 ≤ x) ∧ (x < 1) ⟹
          (prob bern {s | std_unif_cont s ≤ x} = x)
```

CDF_UNIF_CONT

```
∀ x. (prob bern {s | std_unif_cont s ≤ x} =
          (if (x < 0) then 0 else
                 (if (x < 1) then x else 1)))
```

CDF_UNIF_CONT_CONTL

```
∀ x. (0 ≤ x) ∧ (x < 1) ⟹
          ((λx. prob bern {s | std_unif_cont s ≤ x}) contl x)
```

10.2 cdfTheory

This theory contains the formal specification of the CDF along with the formal proofs of its properties mentioned in [14].

10.2.1 Signature

Functions	**Type**
CDF	((*num* → *bool*) → *real*) → *real* → *real*
CDF_in_events_bern	((*num* → *bool*) → *real*) → *real* → *bool*

10.2.2 Definitions

CDF_def

```
⊢def  ∀ f x. CDF f x = prob bern {s | f s ≤ x}
```

CDF_in_events_bern_def

```
⊢def  ∀ f x. CDF_in_events_bern f x = {s | f s ≤ x} IN events bern
```

10.2.3 Theorems

REAL_SUB_ASSOC

```
∀ (a: real) b c. a − b − c = a − (b + c)
```

PROB_COUNTABLE_DECREASING

```
∀ s (f:num −> (num −> bool) −> bool) p.
            (prob_space p) ∧
            (f IN (UNIV −> events p)) ∧
            (f 0 = UNIV) ∧
            (∀n. f (SUC n) SUBSET f n) ∧
            (s = BIGINTER (IMAGE f UNIV)) ⟹
                            prob p o f −−> prob p s
```

PROB_DECREASING_INTER

```
∀ s (f:num −> (num −> bool) −> bool).
            (f IN (UNIV −> events bern)) ∧
            (∀n. f (SUC n) SUBSET f n) ∧
            (s = BIGINTER (IMAGE f UNIV)) ⟹
                            prob bern o f −−> prob bern s
```

CDF_RANGE

```
∀ f x. CDF_in_events_bern f x ⟹
            (0 ≤ CDF f x ∧ CDF f x ≤ 1)
```

CDF_NON_DECREASING

```
∀ f a b. a < b ∧
            (∀x. CDF_in_events_bern f x) ⟹  (CDF f a ≤ CDF f b)
```

CDF_INTERVAL_PROB

```
∀ f a b. a < b ∧
            (∀x. CDF_in_events_bern f x) ⟹
                (prob bern {s | (a < f s) ∧ (f s ≤ b)} =
                                               CDF f b − CDF f a)
```

CDF_AT_POSITIVE_INFINITY

```
∀ f. (∀x. CDF_in_events_bern f x) ⟹
                                (λn. CDF f ((λn. &n) n)) −−> 1
```

CDF_AT_NEGETIVE_INFINITY

```
∀ f. (∀x. CDF_in_events_bern f x) ⟹
                                (λn. CDF f ((λn. ¬&n) n)) −−> 0
```

CDF_CONT_RIGHT

```
∀ f a. (∀x. CDF_in_events_bern f x) ⟹
                                (λn. CDF f ((λn. a +
                                (inv (& (SUC n)))) n)) −−> CDF f a
```

CDF_LIMIT_FROM_LEFT

```
∀ f a. (∀x. CDF_in_events_bern f x) ⟹
                                ((λn. CDF f ((λn. a −
                                (inv (& (SUC n)))) n)) −−>
                                    (prob bern {s | f s < a}))
```

PROB_EQ_SUBSET

```
∀ f a. {s | f s = a} SUBSET
            {s | ((a − inv (& (SUC n))) < f s) ∧ (f s ≤ a)}
```

CONT_CDF_CONT_LEFT

```
∀ f a. (∀x. CDF_in_events_bern f x) ∧
            (∀x. (λx. CDF f x) contl x) ⟹
            ((λn. CDF f (a − (inv (& (SUC n))))) −−> (CDF f a))
```

CONT_CDF_EQ_PROB_BERN_LT

```
∀ f a.(∀x. CDF_in_events_bern f x) ∧
            (∀x. (λx. CDF f x) contl x) ⟹
                (CDF f a = prob bern {s | f s < a})
```

CONT_PROB_BERN_EQ_0

```
∀ f a.(∀x. CDF_in_events_bern f x) ∧
            (∀x. {s | f s = x} IN events bern) ∧
            (∀x. (λx. CDF f x) contl x) ⟹
                (prob bern {s | f s = a} = 0)
```

10.3 itmTheory

This theory contains the formal specification of the inverse function of a CDF, *INV_CDF _FN*, and the formal proof of correctness for the ITM.

10.3.1 Signature

Functions	**Type**
INV_CDF_FN	$(real \rightarrow real) \rightarrow (real \rightarrow real) \rightarrow bool$
IS_CONT_CDF_FN	$(real \rightarrow real) \rightarrow bool$

10.3.2 Definitions

INV_CDF_FN_def

```
⊢def  ∀ (f: real −> real) (g: real −> real). INV_CDF_FN  f g =
                      (∀ x. (g x = 0) ⟹  (x ≤ f (g x))) ∧
                      (∀ x. (g x = 1) ⟹  (f (g x) ≤ x)) ∧
                      (∀ x. (0 < g x ∧ g x < 1) ⟹
                          (f (g x) = x) ∧
                          (∀x. 0 < x ∧ x < 1 ⟹
                                  (g (f x) = x)))
```

IS_CONT_CDF_FN_def

```
⊢def  ∀ (g: real -> real). IS_CONT_CDF_FN g =
                 (∀ a b. a < b ⟹  g a ≤ g b) ∧
                 ((λn. g ((λn. &n) n)) --> 1) ∧
                 ((λn. g ((λn. -&n) n)) --> 0) ∧
                 (∀ x. g contl x)
```

10.3.3 Theorems

CDF_UNIF_CONT_GT0_LT1

```
∀ x. (0 < x) ∧ (x < 1) ⟹
            (prob bern {s | std_unif_cont s ≤ x} = x)
```

CDF_UNIF_CONT_GE0_LE1

```
∀ x. (0 ≤ x) ∧ (x ≤ 1) ⟹
            (prob bern {s | std_unif_cont s ≤ x} = x)
```

STD_UNIF_CONT_NEQ_0_1

```
∀ (a:num -> bool).
            ¬(std_unif_cont a = 1) ∧
            ¬(std_unif_cont a = 0) ⟹
                    ((std_unif_cont a < 1) ∧
                    (0 < std_unif_cont a))
```

FN_MONO_INV_FN_STRICT_MONO

```
∀ f g a b.  (∀a b. a < b ⟹  g a ≤ g b) ∧
                    (∀x. ((0 < x) ∧ (x < 1)) ⟹  (g (f x) = x)) ∧
                    (0 < a) ∧ (a < 1) ∧ (0 < b) ∧ (b < 1) ⟹
                                            ((f a < f b) = (a < b))
```

SET_DIFF_STD_UNIF_CONT_0_1

```
∀ (f:real -> real) x.
            {s | f (std_unif_cont s) ≤ x} IN events bern  ⟹
                (prob bern {s | f (std_unif_cont s) ≤ x} =
                 prob bern (({s | f (std_unif_cont s) ≤ x} DIFF
                             {s | std_unif_cont s = 1}) DIFF
                             {s | std_unif_cont s = 0}))
```

LIM_POS1_NEG0_IMP_GE0_LE1

```
∀ g. IS_CONT_CDF_FN g  ⟹
                (∀x. 0 ≤ g x ∧ g x ≤ 1)
```

CONT_CDF_EXISTS_GT0_LT1

```
∀ g. IS_CONT_CDF_FN g  ⟹
                (∃y. 0 < g y ∧ g y < 1)
```

FN_STD_UNIF_CONT_EQ_EVENTS_BERN

```
∀ f g x. IS_CONT_CDF_FN g ∧ (INV_CDF_FN f g) ⟹
             {s | f (std_unif_cont s) = x} IN events bern
```

FN_STD_UNIF_CONT_EVENTS_BERN

```
∀ f g x. IS_CONT_CDF_FN g ∧ (INV_CDF_FN f g) ⟹
             {s | f (std_unif_cont s) ≤ x} IN events bern
```

ITM_HELPER

```
∀ f g x. IS_CONT_CDF_FN g ∧ (INV_CDF_FN f g) ⟹
             (prob bern  {s | f (std_unif_cont s) ≤ x} =
                 prob bern  {s | std_unif_cont s ≤ g x})
```

ITM

```
∀ f g x. IS_CONT_CDF_FN g ∧ (INV_CDF_FN f g) ⟹
             (prob bern {s | f (std_unif_cont s) ≤ x} = g x)
```

10.4 cont_distTheory

This theory contains the formal proofs of CDF, PMF and measurability properties of four continuous random variables; Uniform, Exponential, Rayleigh and Triangular.

10.4.1 Signature

Functions	**Type**
uniform_rv_def	*real* → *real* → (*num* → *bool*) → *real*
exp_rv_def	*real* → (*num* → *bool*) → *real*
rayleigh_rv_def	*real* → (*num* → *bool*) → *real*
triangular_rv_def	*real* → (*num* → *bool*) → *real*

10.4.2 Definitions

uniform_rv_def

```
⊢def  ∀ a b s. uniform_rv a b s = (b − a) * (std_unif_cont s) + a
```

exp_rv_def

```
⊢def  ∀ l s. exp_rv l s = ¬1/l * ln (1 − (std_unif_cont s))
```

rayleigh_rv_def

```
⊢def  ∀ sig s. rayleigh_rv sig s =
             sig * sqrt ¬2 * ln (1 − (std_unif_cont s)))
```

triangular_rv_def

```
⊢def  ∀ a s. triangular_rv a s =
              a * (1 − sqrt (1 − (std_unif_cont s)))
```

10.4.3 Theorems

INV_CDF_FN_UNIF

```
∀ a b. (a < b) ⟹
           (INV_CDF_FN (λx. (b − a) * x + a)
                      (λx. (if x ≤ a then 0 else
                           (if x < b then (x − a) / (b − a)
                                             else 1))))
```

UNIF_CDF_MONO

```
∀ a b. (a < b) ⟹  (∀c d. (c < d) ⟹
           ((λx. (if x ≤ a then 0 else
               (if x < b then (x − a) / (b − a) else 1))) c ≤
           (λx. (if x ≤ a then 0 else
               (if x < b then (x − a) / (b − a) else 1))) d))
```

UNIF_CONT_HELPER

```
∀ x a b. (a < b) ⟹  ((λx. (x − a) / (b − a)) contl x)
```

UNIF_CDF_CONT

```
∀ x a b. (a < b) ⟹
           ((λx. if x ≤ a then 0 else
               if x < b then (x − a) / (b − a) else 1)) contl x
```

UNIF_CDF_AT_POSINF

```
∀ a b. (a < b) ⟹
           (((λn.(λx. (if x ≤ a then 0 else
               (if x < b then (x − a) / (b − a) else 1)))
                                  ((λn. &n) n)) −−> 1))
```

UNIF_CDF_AT_NEGINF

```
∀ a b. (a < b) ⟹
           (((λn.(λx. (if x ≤ a then 0 else
               (if x < b then (x − a) / (b − a) else 1)))
                                  ((λn. ¬&n) n)) −−> 0))
```

UNIF_CDF_IS_CONT_CDF_FN

```
∀ a b. (a < b) ⟹
           (IS_CONT_CDF_FN (λx. (if x ≤ a then 0 else
                               (if x < b then (x − a) / (b − a)
                                                 else 1))))
```

UNIF_RV_LE_IN_EVENTS_BERN

```
∀ x a b. (a < b) ⟹
           (s | uniform_rv a b s ≤ x IN events bern)
```

UNIF_RV_EQ_IN_EVENTS_BERN

```
∀ x a b. (a < b) ⟹
            (s | uniform_rv a b s = x IN events bern)
```

CDF_UNIF

```
∀ x a b. (a < b) ⟹
            (prob bern {s | uniform_rv a b s ≤ x} =
                                  (if x ≤ a then 0 else
                                  (if x < b then (x − a) / (b − a)
                                   else 1)))
```

PMF_UNIF

```
∀ x a b. (a < b) ⟹
            (prob bern {s | uniform_rv a b s = x} = 0)
```

INV_CDF_FN_EXP

```
∀ l. (0 < l) ⟹
            (INV_CDF_FN (λx. ¬1/l) * ln (1 − x))
                        (λx. if x ≤ 0 then 0 else
                                (1 − exp ¬l * x))))
```

EXP_CDF_MONO

```
∀ l. (0 < l) ⟹  (∀c d. (c < d) ⟹
            ((λx. if x ≤ 0 then 0 else (1 − exp ¬l * x))) c ≤
             (λx. if x ≤ 0 then 0 else (1 − exp ¬l * x))) d))
```

EXP_DIFF_COMPOSITE

```
∀ g m x. ((g diffl m) x ⟹
            ((λx. exp (g x)) diffl (exp (g x) * m)) x)
```

ONE_MINUS_EXP_CONT

```
∀ x l. (0 < l) ⟹
            (λx. 1 − exp ¬l * x)) contl x
```

EXP_CDF_CONT

```
∀ x l. (0 < l) ⟹
            ((λx. (if x ≤ 0 then 0 else
                    1 − exp ¬l * x))) contl x)
```

EXP_CDF_AT_POSINF

```
∀ l. (0 < l) ⟹
            (((λn.(λx. if x ≤ 0 then 0 else
                (1 − exp ¬l * x))) ((λn. &n) n)) −−> 1))
```

EXP_CDF_AT_NEGINF

```
∀ l. (0 < l) ⟹
            (((λn.(λx. if x ≤ 0 then 0 else
                (1 − exp ¬l * x))) ((λn. ¬&n) n)) −−> 0))
```

EXP_CDF_IS_CONT_CDF_FN

```
∀ l. (0 < l) ⟹
          (IS_CONT_CDF_FN (λx. if x ≤ 0 then 0 else
                                 (1 − exp ¬l * x))))
```

EXP_RV_LE_IN_EVENTS_BERN

```
∀ x l. (0 < l) ⟹
          (s | exp_rv l s ≤ x IN events bern)
```

EXP_RV_EQ_IN_EVENTS_BERN

```
∀ x l. (0 < l) ⟹
          (s | exp_rv l s = x IN events bern)
```

CDF_EXP

```
∀ x l. (0 < l) ⟹
          (prob bern {s | exp_rv l s ≤ x} =
                   (if x ≤ 0 then 0 else
                         (1 − exp ¬l * x))))
```

PMF_EXP

```
∀ x l. (0 < l) ⟹
          (prob bern {s | exp_rv l s = x} = 0)
```

INV_CDF_FN_RAYLEIGH

```
∀ sig. (0 < sig) ⟹
          (INV_CDF_FN (λx. sig * sqrt ¬2 * ln (1 − x)))
                   (λx. (if x ≤ 0 then 0 else
                      (1 − exp
                         ¬(x pow 2)/ (2 * (sig pow 2)))))))
```

RAYLEIGH_CDF_MONO

```
∀ sig. (0 < sig) ⟹  (∀c d. (c < d) ⟹
          ((λx. (if x ≤ 0 then 0 else
             (1 − exp ¬(x pow 2)/ (2 * (sig pow 2)))))) c ≤
           (λx. (if x ≤ 0 then 0 else
             (1 − exp ¬(x pow 2)/ (2 * (sig pow 2)))))) d))
```

ONE_MINUS_RAYLEIGH_CONT

```
∀ x sig. (0 < sig) ⟹
          ((λx. 1 − exp
             ¬(x pow 2)/ (2 * (sig pow 2)))) contl x)
```

RAYLEIGH_CDF_CONT

```
∀ x sig. (0 < sig) ⟹
          ((λx. (if x ≤ 0 then 0 else
             (1 − exp
                ¬(x pow 2)/ (2 * (sig pow 2)))))) contl x)
```

POWPOW_PLUS1

```
∀ e. 0 < e ⟹
      (∀n. 1 + & n * e ≤ (1 + e) pow (n * n))
```

SEQ_POWPOW

```
∀ c. 0 < c ∧ c < 1 ⟹
      ((λn. c pow (n*n)) −−> &0)
```

RAYLEIGH_CDF_AT_POSINF

```
∀ sig. (0 < sig) ⟹
      (((λn.(λx. (if x ≤ 0 then 0 else
        (1 − exp ¬(x pow 2)/ (2 * (sig pow 2))))))
                    ((λn. &n) n)) −−> 1))
```

RAYLEIGH_CDF_AT_NEGINF

```
∀ sig. (0 < sig) ⟹
      (((λn.(λx. (if x ≤ 0 then 0 else
        (1 − exp ¬(x pow 2)/ (2 * (sig pow 2))))))
                    ((λn. ¬&n) n)) −−> 0))
```

RAYLEIGH_CDF_IS_CONT_CDF_FN

```
∀ sig. (0 < sig) ⟹
      (IS_CONT_CDF_FN (λx. (if x ≤ 0 then 0 else
                (1 − exp
                ¬(x pow 2)/ (2 * (sig pow 2)))))))
```

RAYLEIGH_RV_LE_IN_EVENTS_BERN

```
∀ x sig. (0 < sig) ⟹
      (s | rayleigh_rv sig s ≤ x IN events bern)
```

RAYLEIGH_RV_EQ_IN_EVENTS_BERN

```
∀ x sig. (0 < sig) ⟹
      (s | rayleigh_rv sig s = x IN events bern)
```

CDF_RAYLEIGH

```
∀ x sig. (0 < sig) ⟹
      (prob bern {s | rayleigh_rv sig s ≤ x} =
             (if x ≤ 0 then 0 else
              (1 − exp
                ¬(x pow 2)/ (2 * (sig pow 2))))))
```

PMF_RAYLEIGH

```
∀ x sig. (0 < sig) ⟹
      (prob bern {s | rayleigh_rv sig s = x} = 0)
```

INV_CDF_FN_TRIANGULAR

```
∀ x a.(0 < a) ⟹
      (INV_CDF_FN (λx. a * (1 − sqrt (1 − x))))
        (λx. if (x ≤ 0) then 0 else
          (if (x < a) then
            (2/a * (x − (x pow 2)/(2 * a))) else 1))
```

TRIANGULAR_CDF_MONO

```
∀ a . (0 < a) ⟹
      (∀c d. (c < d) ⟹
        ((λx. if (x ≤ 0) then 0 else
          (if (x < a) then
            (2/a * (x − (x pow 2)/(2 * a))) else 1)) c ≤
         (λx. if (x ≤ 0) then 0 else
          (if (x < a) then
            (2/a * (x − (x pow 2)/(2 * a))) else 1)) d))
```

TRIANGULAR_CDF_AT_POSINF

```
∀ a. (0 < a) ⟹
      (((λn.(λx. if (x ≤ 0) then 0 else
          (if (x < a) then
            (2/a * (x − (x pow 2)/(2 * a))) else 1))
                      ((λn. &n) n)) −−> 1))
```

TRIANGULAR_CDF_AT_NEGINF

```
∀ a. (0 < a) ⟹
      (((λn.(λx. if (x ≤ 0) then 0 else
          (if (x < a) then
            (2/a * (x − (x pow 2)/(2 * a))) else 1))
                      ((λn. ¬&n) n)) −−> 0))
```

TRIANGULAR_CONT_HELPER

```
∀ x a. (0 < a) ⟹
      (λx. (2/a * (x − (x pow 2)/(2 * a)))) contl x
```

TRIANGULAR_CDF_CONT

```
∀ x a. (0 < a) ⟹
      (λx. if (x ≤ 0) then 0 else
          (if (x < a) then
            (2/a * (x − (x pow 2)/(2 * a))) else 1))
                              contl x
```

TRIANGULAR_CDF_IS_CONT_CDF_FN

```
∀ a. (0 < a) ⟹
      (IS_CONT_CDF_FN (λx. if (x ≤ 0) then 0 else
          (if (x < a) then
            (2/a * (x − (x pow 2)/(2 * a))) else 1)))
```

TRIANGULAR_RV_LE_IN_EVENTS_BERN

```
∀ x a. (0 < a) ⟹
      (s | triangular_rv a s ≤ x IN events bern)
```

TRIANGULAR_RV_EQ_IN_EVENTS_BERN

```
∀ x a. (0 < a) ⟹
      (s | triangular_rv a s = x IN events bern)
```

CDF_TRIANGULAR

```
∀ x a. (0 < a) ⟹
      (prob bern s | triangular_rv a s ≤ x =
        (if (x ≤ 0) then 0 else
          (if (x < a) then
            (2/a * (x − (x pow 2)/(2 * a))) else 1)))
```

PMF_TRIANGULAR

```
∀ x a. (0 < a) ⟹
      (prob bern s | triangular_rv a s = x = 0)
```