Computer Science and Artificial Intelligence Laboratory
Technical Report

MIT-CSAIL-TR-2008-025 May 3, 2008

Random-World Semantics and Syntactic Independence for Expressive Languages

David McAllester, Brian Milch, and Noah D. Goodman

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

Random-World Semantics and Syntactic Independence for Expressive Languages

David McAllester
Toyota Technological Institute at Chicago
Chicago, IL 60637
mcallester@tti-c.org

Brian Milch
MIT CSAIL
Cambridge, MA 02139
milch@csail.mit.edu

Noah D. Goodman
MIT Brain and Cognitive Sciences
Cambridge, MA 02139
ndg@mit.edu

Abstract

We consider three desiderata for a language combining logic and probability: logical expressivity, random-world semantics, and the existence of a useful syntactic condition for probabilistic independence. Achieving these three desiderata simultaneously is nontrivial. Expressivity can be achieved by using a formalism similar to a programming language, but standard approaches to combining programming languages with probabilities sacrifice random-world semantics. Naive approaches to restoring random-world semantics undermine syntactic independence criteria. Our main result is a syntactic independence criterion that holds for a broad class of highly expressive logics under random-world semantics. We explore various examples including Bayesian networks, probabilistic context-free grammars, and an example from Mendelian genetics. Our independence criterion supports a case-factor inference technique that reproduces both variable elimination for BNs and the inside algorithm for PCFGs.

Introduction

We are interested in the combination of logic and probability. In particular we consider the following three desiderata for formal languages which simultaneously have both a logical and a probabilistic semantics.

1. The logical formulas should be expressive. They should support a rich ontology of types such as lists and graphs and should support function composition and recursive definitions.
2. The language should have a random-world semantics. The semantics should be defined by a probability distribution (or measure) on a set of worlds where each logical formula is either true or false in each world, according to the semantics of the underlying logic.
3. The language should have a useful syntactic criteria for probabilistic independence.

Our main technical result is the validity of a set of inference rules for a broad class of expressive probabilistic languages. One of these rules, a consequence of random-world semantics, guarantees that logically equivalent formulas have the same probability. Another is a syntactic criteria for probabilistic independence which allows the program structure to be exploited for inference. Before presenting the main result, however, we elaborate on each of the above desiderata.

First we consider expressivity. We are not satisfied with a language merely as expressive as first order logic. We would like to be able to express, for instance, the notion of reachability — the notion that one given state is reachable from another using given operations. A simple consequence of the compactness theorem is that reachability is not definable in first order logic. More generally, first order logic does not support recursive definitions. Also, even in highly expressive first-order probabilistic languages such as Bayesian logic (Milch *et al.* 2005), there is no straightforward way to define new procedures and use them in several places in a model, or to define distributions over structured data types such as lists or graphs.

To gain sufficient expressivity we turn to functional programming languages, which support recursive definitions as well as a rich ontology of structured data types. We consider languages where the logical formulas are taken to be the Boolean expressions of a programming language, possibly extended with additional forms of quantification. The use of programming languages as a foundation for logical reasoning has a long tradition in the formal methods community (Kaufmann, Manolios, & Moore 2000; Gordon & Melham 1993; Bertot & Castran 2004). The syntactic independence criterion developed here is independent of the particular choice of programming language.

Next we consider various forms of probabilistic semantics. A common way of combining probability with programming is to introduce a random number generator. We can introduce the expression **rand**`()` into the language, and define the language's semantics in terms of a random (stochastic) evaluation process. This is analogous to the use of the standard random number generator in C: each time the evaluator evaluates the expression **rand**`()` it generates a new random number. Random-evaluation semantics is used in Avi Pfeffer's modeling and inference system IBAL (Pfeffer 2001), as well as a number of other probabilistic programming languages (Koller, McAllester, & Pfeffer 1997; Ramsey & Pfeffer 2002; Park, Pfenning, & Thrun 2005).

Random-evaluation semantics is not a random-world semantics. Consider a logical formula of the form $\Phi \wedge \neg\Phi$. Under any random-world semantics this logical formula will have probability zero, as we show below. But under random-evaluation semantics, the two occurrences of the formula Φ can have different values in a single evaluation, since different random choice can be made in a second evaluation of Φ. So the formula $\Phi \wedge \neg\Phi$ can evaluate to true. A related consequence of the lack of random-world semantics is that if one wants to reason about a random function — say, a function `color` mapping balls to colors — one cannot simply use a function in the programming language. Calling `color` on the same ball multiple times would yield different values. To create a random mapping in a random-evaluation language, one must explicitly construct a data structure, such as a list of ball–color pairs.

Next we consider memoization semantics, in which one passes a "name" argument (perhaps an integer or a character string) to the random number generator; we write **rand** (n) where n is a name for the random number. Later calls to the random number generator with the same name return the same number. Memoization guarantees that the same expression evaluates to the same value when it is evaluated a second time. We then get that $\Phi \wedge \neg\Phi$ must evaluate to false. Under memoization semantics we can think of the "world" as being a mapping from names to numbers — a world picks a single random number for each name. This is the approach taken in probabilistic Horn abduction (Poole 1993) and PRISM (Sato & Kameya 2001).

The difficulty with memoization semantics involves the desire for syntactic criterion for independence. Under memoization semantics each expression of the form **rand** (n) can be viewed as an independent random variable — we have one random variable for each name n. Two logical formulas are independent if they involve disjoint sets of random variables; the problem is that it can be difficult to syntactically determine what random variables influence the value of a given formula. This follows because procedures can create names for random variables in arbitrary ways.

Finally we consider a new form of semantics we call number-tree semantics. As in memoization semantics, we add an argument to the random number generator. We still write **rand** (t), but now t is a conceptual data structure called a number tree. A number tree is an infinitely deep, infinitely branching tree in which every node is labeled with a real number in the interval $[0, 1]$. A number tree is a source of random numbers and **rand** (t) simply denotes the number at the root of t. We will show that by working with variables ranging over number trees, it is possible to formulate a syntactic independence criterion. Number-tree semantics satisfies all three of the stated desiderata.

Number-Tree Semantics

We define a number tree to be a tree each node of which is labeled with a real number in the interval $[0, 1]$, and where each node has a countably infinite number of children. The children are indexed by character strings. If t is a number tree and s is a character string, we write $t.s$ to denote the subtree of t reached by going down the branch named by s. The tree $t.s$ is recursively a number tree. The expression **rand** (t) denotes the number labeling the root of the number tree t. We will define a probabilistic semantics where each number in a number tree is selected independently and uniformly from the interval $[0, 1]$.

```
define cloudy(t)
  f_c(t.cloudy)

define f_c(t)
  rand(t) < 0.5

define rain(t)
  f_r(cloudy(t),t.rain)

define f_r(c,t)
  if c then (rand(t) < 0.8)
       else (rand(t) < 0.2)

define sprinkler(t)
  f_s(cloudy(t),t.sprinkler)

define f_s(c,t)
  if c then (rand(t) < 0.1)
       else (rand(t) < 0.5)

define wet(t)
  f_w(rain(t),sprinkler(t),t.wet)

define f_w(r,s,t)
  if r ∨ s then (rand(t) < 0.9)
            else (rand(t) < 0.1)
```

Figure 1: Program for the sprinkler example.

Given the basic notion of a number tree, other details of a particular programming language are fairly arbitrary — number-tree semantics can be used as an extension of any functional programming language. We give a formal treatment for a particular family of languages below. Here we present the ideas of number tree semantics and the desiderata through examples, with programming language features introduced as they are used.

We begin with a simple Bayesian network — the classic "sprinkler" example. In this example, whether the grass is wet (W) depends on whether it has rained recently (R) and whether the sprinkler has been on recently (S). The probabilities of R and S both depend on whether it is cloudy (C). A program for this scenario is shown in Fig. 1. In the figure, the parameter t ranges over number trees and all procedures return Boolean values. The procedures f_c, f_r, f_s and f_w implement conditional probability tables. Each such procedure takes some number of parent values, plus a number tree containing "random" numbers, and returns the value of a node.

In a typed programming language one typically has expressions of type Boolean which we will call logical formulas. Let Φ be a logical formula in which the number tree variable t occurs as the only free variable. Given a value (a number tree) for the variable t, the formula Φ has a well-defined truth value. We let $P_{\mathtt{t}}[\Phi]$ be the probability that when t is assigned to a random tree (with every number selected uniformly in $[0, 1]$) we have that Φ evaluates to true. Given the function definitions in Fig. 1, we can write expressions such as $P_{\mathtt{t}}[\mathtt{wet(t)}]$ for the probability that the grass is wet, or $P_{\mathtt{t}}[\mathtt{sprinkler(t)} \wedge \mathtt{wet(t)}]/P_{\mathtt{t}}[\mathtt{wet(t)}]$ for the conditional probability that the sprinkler was on given that the grass is wet.

Inference Rules

We now present a set of inference rules for a formalism combining logic and probability under number-tree semantics. We begin with an inference rule for case analysis. To formulate this rule, it is convenient to define the set of feasible values for an expression e. First, for any logical formula Φ we define the formula $\exists_t[\Phi]$ to be true if there exists a number tree such that Φ is true when the variable t is interpreted as that tree. The feasible values for an expression e, denoted $V_t[e]$, can then be defined as follows (where v ranges over values specified by the specific language):

$$V_t[e] \quad = \quad \{v: \ \exists_t[e = v]\}$$

We can express the operation of case anaylsis with the following equation.

$$P_{\mathrm{t}}[\Phi] \quad = \quad \sum_{v \in V_t[e]} P_{\mathrm{t}}(\Phi \wedge e = v) \qquad (1)$$

In case analysis one reduces a probability to a sum of simpler probabilities. The choice of the expression e on which to do the case analysis is heuristic.

Logical inference can be used to rewrite logical expressions, replacing a logical formula by an equivalent simpler one. In a random-world semantics these logical equivalencies are respected by the probabilistic aspect of the language. This principle can be expressed as follows:

$$P_{\mathrm{t}}[\Phi] \quad = \quad P_{\mathrm{t}}[\Psi] \ \text{ if } \forall_t[\Phi \Leftrightarrow \Psi] \qquad (2)$$

We now give a syntactic independence criterion that follows from the number-tree semantics. Define a *subtree expression* to be an expression of the form $t.c_1.c_2 \ldots c_n$ where each c_i is a character string constant: for example, $t.\mathtt{left}.\mathtt{right}$. Now consider two subtree expressions $t.\alpha$ and $t.\beta$ where α and β are each sequences of character string constants. We say that $t.\alpha$ is syntactically independent of $t.\beta$ provided that neither α nor β is a prefix of the other. For example, $t.\mathtt{left}.\mathtt{right}$ and $t.\mathtt{left}.\mathtt{left}$ are syntactically independent but $t.\mathtt{left}.\mathtt{right}$ and $t.\mathtt{left}$ are not. Syntactically independent subtree expressions denote disjoint substrees. The number labels in disjoint subtrees are independent. Now consider two logical formulas Φ ad Ψ and a number tree variable t.

Definition 1. *Two formulas Φ and Ψ are* syntactically independent *with respect to number tree variable t if for every pair of an occurrence of t in Φ and an occurrence of t in Ψ we have that the occurrence of t in Φ occurs inside a subtree expression $t.\alpha$ and the occurrence of t in Ψ occurs inside a subtree expression $t.\beta$ where $t.\alpha$ and $t.\beta$ are syntactically independent.*

The main technical contribution of this paper is number tree semantics and the observation that under number tree semantics, the following factorization rule holds:

$$P_{\mathrm{t}}[\Phi \wedge \Psi] = P_{\mathrm{t}}[\Phi] P_{\mathrm{t}}[\Psi] \ \text{ if } \Phi \text{ and } \Psi \text{ are synt. ind.} \qquad (3)$$

Finally, we have the following contraction rule where $\Phi[t.c]$ is a logical formula and t is a number tree varaible such that every occurrence of t in $\Phi[t.c]$ occurs inside the subtree expression $t.c$.

$$P_{\mathrm{t}}[\Phi[t.c]] = P_{\mathrm{t}}[\Phi[t]] \qquad (4)$$

Inference in Bayesian Networks

Here we show that the inference rules of the previous section are sufficient for the analysis of Bayesian networks. We will analyze the sprinkler example, the generalization to arbitrary Bayesian networks is straightforward. Consider evaluating the probability $P_{\mathrm{t}}[\mathtt{wet(t)\ =\ w}]$. By logical equivalence (2) we can replace each random variable (each node of the network) by the appropriate call to a conditional probability table procedure yielding:

$$P_{\mathrm{t}}\left[f_{\mathrm{w}}\left(\begin{array}{l} f_{\mathrm{r}}(f_{\mathrm{c}}(\mathtt{t.cloudy}), \mathtt{t.rain}), \\ f_{\mathrm{s}}(f_{\mathrm{c}}(\mathtt{t.cloudy}), \mathtt{t.sprinkler}), \\ \mathtt{t.wet} \end{array} \right) = w \right]$$

We can now do a case anlysis (1) on the unobserved nodes (all nodes other than the observed node `wet(t)`) yielding:

$$\sum_{r,s,c} P_{\mathrm{t}} \left[\begin{array}{l} f_{\mathrm{w}}\left(\begin{array}{l} f_{\mathrm{r}}(f_{\mathrm{c}}(\mathtt{t.cloudy}), \mathtt{t.rain}), \\ f_{\mathrm{s}}(f_{\mathrm{c}}(\mathtt{t.cloudy}), \mathtt{t.sprinkler}), \\ \mathtt{t.wet} \end{array} \right) = w \\ \wedge f_{\mathrm{r}}(f_{\mathrm{c}}(\mathtt{t.cloudy}), \mathtt{t.rain}) = r \\ \wedge f_{\mathrm{s}}(f_{\mathrm{c}}(\mathtt{t.cloudy}), \mathtt{t.sprinkler}) = s \\ \wedge f_{\mathrm{c}}(\mathtt{t.cloudy}), \mathtt{t.sprinkler}) = c \end{array} \right]$$

Now by logical equivalence (2) this is equivalent to:

$$\sum_{r,s,c} P_{\mathrm{t}} \left[\begin{array}{l} f_{\mathrm{w}}(r, s, \mathtt{t.wet}) = w \\ \wedge f_{\mathrm{r}}(c, \mathtt{t.rain}) = r \\ \wedge f_{\mathrm{s}}(c, \mathtt{t.sprinkler}) = s \\ \wedge f_{\mathrm{c}}(\mathtt{t.cloudy}) = s \end{array} \right]$$

Now by syntactic independence (3) we can factor this as:

$$\sum_{r,s,c} \left\{ \begin{array}{l} P_{\mathrm{t}}[f_{\mathrm{w}}(r, s, \mathtt{t.wet}) = w] \\ P_{\mathrm{t}}[f_{\mathrm{r}}(c, \mathtt{t.rain}) = r] \\ P_{\mathrm{t}}[f_{\mathrm{s}}(c, \mathtt{t.sprinkler}) = s] \\ P_{\mathrm{t}}[f_{\mathrm{c}}(\mathtt{t.cloudy}) = c] \end{array} \right.$$

Finally, by contraction (4) this can be rewritten as:

$$\sum_{r,s,c} \left\{ \begin{array}{l} P_{\mathrm{t}}[f_{\mathrm{w}}(r, s, \mathtt{t}) = w] \\ P_{\mathrm{t}}[f_{\mathrm{r}}(c, \mathtt{t}) = r] \\ P_{\mathrm{t}}[f_{\mathrm{s}}(c, \mathtt{t}) = s] \\ P_{\mathrm{t}}[f_{\mathrm{c}}(\mathtt{t}) = c] \end{array} \right.$$

This is the standard sum of product expression for the probability of evidence in a Bayesian network. Standard algorithms can be used to evaluate this expression.

A Mendelian Genetics Model

Fig. 2 shows a program for Mendelian genetics. Each individual has a genotype, represented as a mapping from loci (places where genes are located) to pairs of alleles (versions of a gene). The variable t ranges over number trees. In this example the programming language is taken to include data structures build with data constructors where `foo[x,y]` denotes the a data structure with top level tag `foo` and which includes the data structures x and y as parts. the expression $\mathbf{match}(v, p, b_1, b_2)$ matches the value v against the pattern p and, if the match is successful, evaluate b_1 under the variable bindings generated by the match, and other wise returns the value of b_2. In this code data structures are sometimes used rather than character strings to select a child of a given

```
define genotype(x, t)
  match(x, child[id, f, m],
        mate(genotype(f, t), genotype(m, t),
             t.conception[x]),
        genotypePrior(t.origin[x]))

define mate(g1, g2, t)
  hpair(meiosis(g1, t.left),
        meiosis(g2, t.right))

define meiosis(g, t)
  lambda loc
    match(g(loc), pair[a, b]
          (if rand(t.loc) < 0.5 then a else b),
          error[])

define hpair(f,g)
  lambda x
    pair[f(x),g(x)]
```

Figure 2: Program for Mendelian genetics.

number tree. We assume that there is some standard invertable way of coercing data structures to character strings (such as a print method). Individuals with known parents are represented as data structures of the form child$[i, f, m]$, where i is a unique identifier, f is the individual's father, and m is the individual's mother. Individuals with unknown parents can be represented using any other data structures. Thus, we might define a family tree as:

```
let steve = founder['steve']
let jane = founder['jane']
let alice = child['alice', steve, jane]
let phil = child['phil', steve, jane]
let jim = founder['jim']
let bob = child['bob', jim, alice]
```

The definition of the `genotype` function in Fig. 2 says that if the individual `x`, has the form `child[id, f, m]`, then his genotype is generated by the `mate` procedure on the parent genotypes, using the random numbers in the subtree `t.conception[x]`. Otherwise, his genotype is generated by `genotypePrior` using the subtree `t.origin[x]`. We omit the specification of any particular prior. The `mate` procedure performs meiosis—a process that chooses one allele from the pair at each locus—on both parent genotypes, using separate sets of random numbers. The result of meiosis is a haplotype: a function mapping each locus to a single allele. The paternal and maternal haplotypes are combined into a genotype by the deterministic function `hpair`.

The haplotype returned by `meiosis` is defined by a lambda-expression that takes a locus `loc`, interprets the genotype value `g(loc)` as a pair `pair[a, b]`, and returns `a` or `b` with equal probability. A separate random number `t.loc` is used for each locus. Note that becuse `t.loc` has a fixed value, the resulting haplotype function will return a fixed allele if it is invoked multiple times on the same locus.

```
define CFTfrom(x, t)
  case ProdFrom(x, t.center) of
    pair[y,z] :
      node[x, CFTfrom(y, t.left),
           CFTfrom(z, t.right)]
    terminal[a] : terminal[a]

define CFT(t) CFTfrom(s[], t)

define yield(z)
  case z of
    node[x, zleft, zright] :
      append(yield(zleft), yield(zright))
    terminal[a] : list(a)

define CFstring(t) yield(CFT(t))
```

Figure 3: Program that uses a PCFG to define a distribution over trees.

Probabilistic Context-Free Grammars

The program in Fig. 3 defines a procedure `CFTFrom` that takes a nonterminal symbol from the PCFG and a number tree, and returns a syntax tree generated by the PCFG using the random numbers in the number tree. This is done in such a way that when the number tree is generated at random in the standard way we get the probability distribution on output trees appropriate for the given PCFG. The procedure `ProdFrom` takes a nonterminal and a number tree and returns the right hand side of a production from the given nonterminal. We assume that the probability distribution over the value of ProdFrom(x, t) is determined by the parameters of a given PCFG. The expression **case** e of $p_1 : b_1$ $p_2 : b_2$ is an abbreviation for **match** $(e, p_1, b_1,$ **match** $(e, p_2, b_2,$ error[])).

Given the procedure `ProdFrom`, we can define the procedure `CFT` (for context-free tree) shown in Fig. 3. The procedure makes it clear that the left and right subtrees are independent with respect to the randomness introduced by the tree t. This would be true even if the left and right subtrees were computed by arbitrary procedures, provided that the left and right subtrees were computed from the randomness in `t.left` and `t.right` respectively.

The procedure `yield` in Fig. 3 computes the string yielded by a given syntax tree. Putting everything together, the procedure `CFString` returns the yield of a randomly generated tree.

The Inside Algorithm

We now show that our inference rules are sufficient to derive the inside algorithm. Suppose that we want to compute a probability of the following form where s is a given string of terminal symbols.

$$P_{\mathrm{t}}[\texttt{yield(CFTfrom(X[],t))=}s]$$

We can first apply case analysis (1) on `ProdFrom(X[],t.center[])`. For `length(s)>1` the resulting case probability is zero unless the production yields a pair of the form `pair[`y`, `z`]`, where y and z are nonterminal symbols. For the pair cases we are left with computing probabilities of the following form.

$$P_{\mathtt{t}}\left[\begin{array}{l}\mathtt{yield(CFTfrom(X[],t))=}s\\ \wedge\mathtt{ProdFrom(X[],t.center[])=pair[}y\mathtt{,}\ z\mathtt{]}\end{array}\right]$$

Next we case on the value of `length(yield(CFTFrom(`y`,t.left[])))`. This gives a probability of the following form.

$$P_{\mathtt{t}}\left[\begin{array}{l}\mathtt{yield(CFTfrom(X[],t))=}s\\ \wedge\mathtt{ProdFrom(X[],t.center[])=pair[}y\mathtt{,}\ z\mathtt{]}\\ \wedge\mathtt{Length(CFTfrom(}y\mathtt{,t.left[]))=}k\end{array}\right]$$

Now by logical equivalence (2) this is equivalent to a probability of the following form.

$$P_{\mathtt{t}}\left[\begin{array}{l}\mathtt{ProdFrom(X[],t.center[])=pair[}y\mathtt{,}\ z\mathtt{]}\\ \wedge\mathtt{yield(CFTfrom(}y\mathtt{,t.left[]))=}s_1\\ \wedge\mathtt{yield(CFTfrom(}z\mathtt{,t.right[]))=}s_2\end{array}\right]$$

Here s_1 consists of the first k symbols in s, and s_2 is the remainder of s. We can now use syntactic independence (3) and contraction (4) to reduce this to a product of probabilities two of which are recursively of the original form. Dynamic programming on the probability problems of the original form yields the inside algorithm.

Formal Treatment

We start our formal discussion by defining number trees, and a uniform measure on number trees. Let C be the set of all finite character strings. A *node* s is an element of C^*, the set of all finite sequences of finite character strings. Note that C^* can naturally be seen as a tree, where the empty sequence $\varnothing$ is the root node, and the parent of each non-root node $(c_1, \ldots, c_n, c_{n+1})$ is the node $(c_1, \ldots, c_n)$.

A *number tree* $\mathcal{T}$ is a function from C^* to $[0, 1]$. We use T to denote the set of all number trees, $[0, 1]^{C^*}$. For any number tree $\mathcal{T}$ and character string c, the *c-subtree* of $\mathcal{T}$, denoted $\mathcal{T}.c$, is that number tree $\mathcal{T}'$ such that for each node $s = (c_1, \ldots, c_n)$, $\mathcal{T}'(s) = \mathcal{T}((c, c_1, \ldots, c_n))$.

As an event space on number trees, we use the product σ-algebra $\mathcal{B}^{C^*}$, where $\mathcal{B}$ is the Borel σ-algebra on $[0, 1]$. We define the measure P_{unif} on number trees $\mathcal{T}$ such that for each node s, the random variable $\mathcal{T}(s)$ has a uniform distribution on $[0, 1]$, and all these random variables are mutually independent. The existence and uniqueness of this P_{unif} follow from Kolmogorov's extension theorem.

We now consider a particular formal language. Formally specifying a programming language can be tedious. For expediency we consider the simply-typed lambda calculus with constants. We have the following grammar for types and terms respectively.

$$\begin{array}{rcl}\tau & ::= & C \mid \tau_1 \rightarrow \tau_2\\ e & ::= & c \mid x \mid e_1(e_2) \mid \lambda\, x\, e\end{array}$$

Here C denotes a type constant, c denotes a term constant, and x denotes a term variable. Here we use a somewhat nonstandard formulation where we assume that each term constant and term variable is associated with a fixed type which is part of its syntax. We let τ_c be the type associated with the term constant c and τ_x be the type associated with the variable x. We have the standard notion of free and bound variable occurrence. A term with no free variables is called closed. Each type expression τ denotes a set $[\tau]$. We assume each type constant C is associated with a specified set $[C]$. We take $[\tau_1 \rightarrow \tau_2]$ to be the set of all functions from $[\tau_1]$ to $[\tau_2]$. We are only interested in well-typed exressions. We write $\vdash e : \tau$ to mean that e is well-typed with type τ. We have $\vdash c : \tau_c$ and $\vdash x : \tau_x$. We also have $\vdash e_1(e_2) : \sigma$ provided $\vdash e_1 : (\tau \rightarrow \sigma)$ and $\vdash e_2 : \tau$, and $\vdash (\lambda\, x\, e) : \tau_x \rightarrow \sigma$ provided $\vdash e : \sigma$.

We assume that each term constant c is associated with a value $[c] \in [\tau_c]$. We define a type-respecting variable interpretation to be a mapping α on variables with the property that for any variable x we have $\alpha(x) \in [\tau_x]$. We write $\alpha[x := v]$ for the variable interpretation that is identical to α except that it maps x to v. The semantics has the property that if $\vdash e : \tau$ then then for any type-respecting variable interpretation α we have $[e]^\alpha \in [\tau]$. This property follows immediately by induction on expressions under the following definition of the value of expressions.

$$\begin{array}{rcl}[c]^\alpha & = & [c]\\ [x]^\alpha & = & \alpha(x)\\ [e_1(e_2)]^\alpha & = & [e_1]^\alpha([e_2]^\alpha)\\ [\lambda x\, e]^\alpha & = & \text{the function } v \mapsto [e]^{\alpha[x:=v]} \text{ for } v \in [\tau_x]\end{array}$$

Multi-argument functions can be represented by Currying — we represent a function $f : \tau_1 \times \tau_2 \rightarrow \sigma$ by a Curried function $\tau_1 \rightarrow (\tau_2 \rightarrow \sigma)$. We can take `let` x `be` e_1 `in` e_2 to be an abbreviation for $(\lambda\ x\ e_2)(e_1)$ and we can represent a nonrecursive procedure definition $\mathbf{define}\ f(x)\ e$ by `let` f `be` $(\lambda\ x\ e)$ `in` u where u is a program using the defined procedure f. For a closed term e we write $[e]$ for $[e]^\alpha$ where α is arbitrary.

The expressive power of the simply-typed lambda calculus depends on the set of constants included in the language. For concreteness we consider a particular language defined by a particular set of constants. We take the type constants to consist of $string$ for the set of character stings, $node$ for the set of lists of character strings, $real$ for the set of real numbers, and $Bool$ for the set containing "true" and "false". The term constants include real number constants and character string constants. The constants also include the comparison predicates $<$ and $=$ on real numbers and the Boolean operations such as disjunction and negation. They also include `nil`, `cons`, `car`, and `cdr` for making and manipulating nodes (lists of character strings). Also, for each type τ we have the conditional operator $\mathtt{if}_\tau : Bool \times \tau \times \tau \rightarrow \tau$. This allows us to write conditional expressions at any type. A number tree is a function from nodes to reals — the type $tree$ is taken to be an abbreviation for $node \rightarrow real$. We can represent $\mathbf{rand}(t)$ by $t(\mathtt{nil})$ and the subtree operation, as in $t.c$, can be defined in terms of node operations and lambda abstraction. We take this particular set of constants for the simply typed lambda calculus to define the languge $\mathcal{L}_0$. Of course we could consider other languages with more constants and richer type systems.

Our first formal result is the following.

Theorem 1. *If Q is a closed expression of $\mathcal{L}_0$ of type $tree \rightarrow Bool$ then $[Q]$ is a measurable predicate on trees, i.e., the set of $\mathcal{T} \in T$ such that $[Q]^{\alpha}(\mathcal{T})$ is true is a measurable set of trees under the measure defined above.*

Proof Sketch: At an intuitive level, this theorem follows from the fact that the simply-typed lamdba calculus is strongly normalizing. We can compute the value of $Q(\mathcal{T})$ using only a finite number of comparison operations on real numbers. This implies that the set of $\mathcal{T} \in T$ satisfying Q can be written as a union of sets each of which is defined by a finite number of interval restrictions at a finite number of nodes. Without loss of generality we can assume that the interval restrictions have rational endpoints — an interval can be written as a union of intervals with rational endpoints. There are only countably many sets definable by rational interval restrictions at a finite number of nodes. Therefore the set of $\mathcal{T}$ satisfying Q can be written as a countable union of measurable sets and is therefore measurable. $\square$

Now let t be a variable of type $tree$ and let Φ be a Boolean expression of $\mathcal{L}_0$ whose only free variable is t. Given the above measurability theorem, we can define $P_t[\Phi]$ to be the probability that Φ is true for a random tree t, which is by definition the measure of the set of trees $\mathcal{T}$ satisfying the predicate $[\lambda t\ \Phi]$. We can also define $\exists_t[\Phi]$ to be true if there exists a tree $\mathcal{T}$ satisfying the predicate $[\lambda t\ \Phi]$ and we can use $\forall_t[\Phi]$ as an abbreviation for $\neg\exists_t[\neg\Phi]$.

We now formally verify the rules of inference stated earlier. For the case-analysis rule (1) we require that e also has the property that its only free variable is t. Here we also assume that e is Boolean, although this assumption is easily relaxed in languages that support, say, integers and integer arithmetic. For e Boolean the case-analysis rule becomes the following:

$$P_t[\Phi] = P_t[\Phi \wedge \Psi] + P_t[\Phi \wedge \neg\Psi]$$

More generally, the case-analysis rule (1) is meaningful for $\vdash e : \tau$ where the language contains a constant denoting each element of the set $[\tau]$. The other inference rules hold as stated.

Theorem 2. *Inference rules (2), (3) and (4) are valid for any Boolean expressions Φ and Ψ of $\mathcal{L}_0$ whose only free variable is the tree variable t.*

A natural question is how expressive can one make the language while preserving the validity of the above theorems. There is no problem in adding standard data types such as records or structures. If we allow recursion then we need to be careful about termination. Of course recursive functions need not terminate in general. However, if we are careful to only write terminating recursions then it seems that the measurability theorem still holds — we can compute the value of a predicate on trees using only a finite number of comparisons. It also seems likely that the measurability theorem remains true if we allow recursions that need not terminate on all inputs but still terminate with probability one. Another natural extension is to introduce quantification into the langauge. For any type τ we can add a constant $\exists_\tau : (\tau \rightarrow Bool) \rightarrow Bool$ where have that $\exists_\tau(Q)$ is true if there exists a value v in $[\tau]$ such that Q is true of v. We can also allow probabilities to be terms in the langauge. We can add a constant $P : (tree \rightarrow Bool) \rightarrow real$ where $P(Q)$ denotes the probability that Q is true of a random tree. This would allow probability expressions to appear inside the statements used in probability expressions. We do not know whether the measurability theorem holds under (some formulation of) these more extreme extensions. If the measurability theorem fails we can, as a last resort, require that the programmer be careful to only construct probability expressions for measurable properties.

Conclusion

We have developed a particular approach to the combination of logic and probability yielding certain desirable inference rules. These inference rules exhibit both random-world semantics (2) and a syntactic independence principle (3). The rules apply to a formally defined set of highly expressive logical formulas — the expressions of type Boolean in a rich typed lambda calculus. This approach to combining logic and probability, based on number-tree semantics, holds over a wide variety of formal lambda calculi and programming languages. We believe that our inference rules provide a new and more tractable foundation for automated reasoning about the rich class of stochastic models definable by stochastic programs.

References

Bertot, Y., and Castran, P. 2004. *Interactive Theorem Proving and Program Development; Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer.

Gordon, M., and Melham, T. 1993. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press.

Kaufmann, M.; Manolios, P.; and Moore, J. S. 2000. *Computer-Aided Reasoning: An Approach*. Springer.

Koller, D.; McAllester, D. A.; and Pfeffer, A. 1997. Effective Bayesian inference for stochastic programs. In *Proc. 14th AAAI*, 740–747.

Milch, B.; Marthi, B.; Russell, S.; Sontag, D.; Ong, D. L.; and Kolobov, A. 2005. BLOG: Probabilistic models with unknown objects. In *Proc. 19th IJCAI*, 1352–1359.

Park, S.; Pfenning, F.; and Thrun, S. 2005. A probabilistic language based upon sampling functions. In *Proc. 32nd POPL*, 171–182.

Pfeffer, A. 2001. IBAL: A probabilistic rational programming language. In *Proc. 17th IJCAI*, 733–740.

Poole, D. 1993. Probabilistic Horn abduction and Bayesian networks. *Artificial Intelligence* 64(1):81–129.

Ramsey, N., and Pfeffer, A. 2002. Stochastic lambda calculus and monads of probability distributions. In *Proc. 29th POPL*, 154–165.

Sato, T., and Kameya, Y. 2001. Parameter learning of logic programs for symbolic–statistical modeling. *JAIR* 15:391–454.