A Stochastic λ-Calculus

Content Areas: probabilistic reasoning, knowledge representation, causality
Tracking Number: 775

Abstract

There is an increasing interest within the research community in the design and use of recursive probability models. Although there still remains much concern about computational complexity costs, several research groups are developing recursive stochastic languages. We have developed an extension to the traditional λ-calculus as a framework for families of Turing complete stochastic languages. We have also developed a class of exact inference algorithms based on the traditional reductions in the λ-calculus. We further propose that using the deBruijn notation (a λ-calculus notation with nameless dummies) supports effective caching in such systems (caching being an essential component of efficient computation). Finally, our extension to the λ-calculus offers a foundation and general theory for the construction of recursive stochastic modeling languages.

1. Introduction

The limitations of flat Bayesian Networks (BNs) using simple random variables have been widely noted by researchers [Xiang et al., 1993; Laskey and Mahoney, 1997]. These limitations have motivated a variety of recent research projects in hierarchical and composable Bayesian models [Koller and Pfeffer, 1997; Koller and Pfeffer, 1998; Laskey and Mahoney, 1997; Pfeffer et al., 1999; Xiang et al., 2000]. Most of these new Bayesian modeling formalisms support model decomposition, often based on an object-oriented approach. Although these approaches provide more expressive and/or succinct representational frameworks, few of these change the class of models that may be represented.

Recent research has addressed this issue. One example is the functional stochastic modeling language proposed by [Koller et al., 1997]. Their language is Turing complete, allowing the representation of a much broader class of models. [Pless et al., 2000] extends and refines this proposed framework to one which is more object-oriented and which allows hierarchical encapsulation of models. Both languages provide the ability to use functions to represent general stochastic relationships. They both also use lazy evaluation to allow computation over potentially infinite distributions. Pfeffer [2000] and Pfeffer and Koller [2000] have also proposed a Turing complete framework based on approximate inference.

What all these approaches have in common is the development of recursive models that unify inference in Bayesian Networks with more complex models such as stochastic context free grammars. The result aims at allowing the construction and inference in novel Bayesian models. All of these depend on caching of partial results for efficiency purposes, just as efficient inference in Bayesian Networks requires the storage of intermediate values.

In this present paper we offer an extension of the traditional λ-calculus as a foundation for building Turing complete stochastic modeling languages. We have also developed a class of exact stochastic inference algorithms based on the traditional reductions in the λ-calculus. We further propose the use of deBruijn [1972] notation to support effective caching mechanisms for efficient computation. As noted above, caching offers an important technique for support of efficient inference in stochastic networks.

As a final note, other recent research has viewed stochastic modeling in terms of stochastic functions [Pearl, 2000; Koller, and Pfeffer, 1997]. For example, Pearl's [2000] recent book constructs a formalism for "causality" in terms of stochastic functions. We have expanded these ideas to offer a formal structure for such modeling, based on an extension of the λ-calculus, in which the stochastic functions themselves become first class objects.

2. The Extended λ-Calculus Formalism

We now construct a formal grammar reflecting our extension of the λ-calculus to describe stochastic distributions. The goal of this effort is to propose an extended form that also supports an inference algorithm as a set of standard transformations and reductions of λ-calculus forms. Thus, inference in our modeling language is equivalent to finding normal forms in the λ-calculus. We also enhance our language through the use of deBruijn notation [deBruijn, 1972]. This notation replaces arbitrarily chosen variable names with uniquely determined numbers. As a result all expressions that are α-equivalent in standard notation are identical under deBruijn notation. This is very useful in both constructing distributions as well as in caching partial results.

2.1 Syntax

We next present a pseudo-BNF grammar to describe our stochastic extension of the traditional λ-calculus:

<expr> ::= <var> | <λ> | <application> | <distribution>
<var> ::= <integer>
<λ> ::= (λ <expr>)
<application> ::= (<expr>$_1$ <expr>$_2$)
<distribution> ::= $\sum_i$ <expr>$_i$: <p>$_i$
p ∈ (0, 1]

Thus, our stochastic λ-calculus contains the same elements as standard λ-calculus: variables, λ-abstraction, and function application. In addition, in our stochastic λ-calculus, it is legal to have an expression which is itself a distribution of expressions.

When using deBruijn notation, we denote a variable by a number. This number indicates how many λs one must go out to find the one λ to which that variable is bound. We denote a λ-abstraction in this form (λ e) where e is some legal expression. For example (λ 1) represents the identity function. In λ-calculus, boolean values are often represented by functions that take two arguments. True returns the first one, and false the second. In this notation true becomes (λ (λ 2)) and false is (λ (λ 1)), or in an abbreviated form (λλ 2) and (λλ 1) respectively.

For a further example we can use deBruijn notation to describe the S operator from combinatory logic. The S operator may be described by the rule Sxyz = (xz)(yz) which is equivalent to the standard λ term (λxλyλz.(xz)(yz)). In deBruijn notation, this becomes: (λλλ (3 1)(2 1)).

Function application is as one might expect: We have (e_1 e_2), where e_1 is an expression whose value will be applied as a function call on e_2, where e_2 must also be a valid expression. We describe distributions as a set of expressions annotated with probabilities. An example would be a distribution that is 60% true and 40% false. Using the representation for boolean values given above, the resulting expression would be: {(λλ 2): 0.6, (λλ 1): 0.4}. Note that we use a summation notation in our BNF specification. The set notation is convenient for denoting a particular distribution, while the summation notation is better for expressing general rules and algorithms .

2.2 Semantics

We next develop a specification for the semantics of our language. For expressions that do not contain distributions, the semantics (like the syntax) of the language is the same as that of the normal λ-calculus. We have extended the this semantics to handle distributions.

A distribution may be thought of as a variable whose value will be determined randomly. It can take on the value of any element of its set with a probability given by the annotation for that element. For example, if T denotes true as represented above, and F represents false, the distribution {T: 0.6, F: 0.4} represents the distribution over true and false with probability 0.6 and 0.4 respectively.

A distribution applied to an expression is viewed as equivalent to the distribution of each element of the distribution applied to the expression, weighted by the annotated probability. An expression applied to a distribution is likewise the distribution of the expression applied to each element of the distribution annotated by the corresponding probability. Note that in both these situations, when such a distribution is formed it may be necessary to combine identical terms by adding the annotated probabilities.

In all other situations an application of a function to an expression follows the standard substitution rules for the λ-calculus with only one exception: The substitution cannot be applied to a general expression unless it is known that the expression is not reducible to a distribution with more than one term. For example, an expression of the form ((λ e_1) (λ e_2)) can always be reduced to an equivalent expression by substituting e_2 into e_1 because (λ e_2) is not reducible. We describe this situation formally with our presentation of the reductions in the next section on stochastic inference.

There is an important implication of the above semantics. Every application of a function whose body includes a distribution causes an independent sampling of that distribution. There is no correlation between these samples. On the other hand, a function applied to a distribution induces a complete correlation between instances of the bound variables in the body of the function.

For example, using the symbols T and F as described earlier, we produce two similar expressions. The first version, (λ 1 F 1){T: 0.6, F: 0.4}, demonstrates the induced correlations. This expression is equivalent to F (false). This expression is always false because the two 1's in the expression are completely correlated (see the discussion of the inference reductions below for a more formal demonstration). Now to construct the second version let G = (λ {T: 0.6, F: 0.4}). Thus G applied to anything produces the distribution {T: 0.6, F: 0.4}. So the second version ((G T) F (G T)) looks similar to the first one in that they both look equivalent to ({T: 0.6, F: 0.4} F {T: 0.6, F: 0.4}). The second one is equivalent because separate calls to the same function produce independent distributions. The first one is not due to the induced correlation.

Finally, it should be noted that we can express Bayesian Networks and many other more complex stochastic models, including Hidden Markov Models, with our language. As the language is Turing complete, it can represent everything

that other Turing complete languages can. For illustration, we next show how to represent the traditional Bayesian Network in our stochastic λ-calculus.

2.3 An Example: Representing Bayesian Networks

To express a BN, we first construct a basic expression for each variable in the network. These expressions must then be combined to form an expression for a query. At first we just show the process for a query with no evidence. The technique for adding evidence will be shown later. A basic expression for a variable is simply a stochastic function of its parents.

To then form an expression for the query, one must form each variable in turn by passing the distribution for its parents in as arguments. When a variable has more than one child, an abstraction must be formed to bind its value to be passed to each child separately.

Our example BN has three Boolean variables: A, B, and C. Assume A is true with probability of 0.5. If A is true, then B is always true, otherwise B is true with probability of 0.2. Finally, C is true when either A or B is true. Any conditional probability table can be expressed in this way, but the structured ones given in this example yield more terse expressions. The basic expressions (shown in both standard and deBruijn notation) are shown below:

A = {T: 0.5, F: 0.5}
B = (λA.(A T {T: 0.2, F: 0.8})) = (λ 1 T {T: 0.2, F: 0.8})
C = (λAλB.(A T B)) = (λλ 2 T 1)

The complete expression for the probability distribution for C is then ((λ C 1 (B 1)) A). One can use this to express the conditional probability distribution that A is true given that C is true: ((λ (C 1 (B 1)) 1 N) A) where N is an arbitrary term (not equivalent to T or F) that denotes the case that is conditioned away. To infer this probability distribution, one can use the reductions (defined below) to get to a normal form. This will be a distribution over T, F, and N, with the subsequent marginalizing away of N.

In general, to express evidence, one can create a new node in the BN with three states. One state is that the evidence is false, the second is the evidence and the variable of interest are true, and the third represents the evidence is true and the variable of interest is false. One can then get the distribution for the variable of interest by marginalizing away the state representing the evidence being false. The extension to non boolean variables of interest is straightforward.

Of course, a language with functions as first class objects can express more than Bayesian Networks. It is capable of expressing the same set of stochastic models as the earlier Turing complete modeling languages proposed by [Koller et al., 1997; Pless et al., 2000; Pfeffer, 2000; Pfeffer and Koller, 2000]. Any of those languages could be implemented as a layer on top of our stochastic λ-calculus. In [Pless et al., 2000] the modeling language is presented in terms of an outer language for the user which is then transformed into an inner language appropriate for inference. Our stochastic λ-calculus could also be used as a compiled form for a more user friendly representation.

3. Stochastic Inference through l-Reductions

We next describe exact stochastic inference through the traditional methodology of the λ-calculus, a set of λ-reductions. In addition to the β and η reductions, we also define a new type: γ reductions.

β: $((\lambda\, e_1)\ e_2) \rightarrow \text{substitute}(e_1, e_2)$
γ_L: $((\sum_i f_i: p_i)\ e) \rightarrow \sum_i (f_i\ e): p_i$
γ_R: $(f\ \sum_i e_i: p_i) \rightarrow \sum_i (f\ e_i): p_i$
η: $(\lambda\, (e\ 1)) \rightarrow e$

We have defined β reductions in a fashion similar to standard λ-calculus. Since we are using deBruijn notation, α transformations become unnecessary (as there are no arbitrary dummy variable names). β and η reductions are similar to their conventional counterparts (see deBruijn [1972] for restrictions on when they may be applied). The one in our case difference is that β reductions are more restricted in that expressions that are reducible to distributions cannot be substituted. In addition to those two standard reductions we define two additional reductions that we term γ_L and γ_R. The γ reductions are based on the fact that function application and distributions distribute.

One important advantage of using deBruijn notation is the ability to reuse expressions when performing substitutions. We next present a simple algorithm for substitutions when e_2 is a closed expression:

```
level(expr) = case expr
    var → expr
    (λ e) → max(level(e) – 1, 0)
    (e1 e2) → max(level(e1), level(e2))
    ∑i ei: pi → maxi(level(ei))

substitute((λ e1), e2) = substitute(e1, e2, 1)

substitute(expr, a, L) =  if level(expr) < L then expr
    else case expr
        var → a
        (λ e) → (λsubstitute(e, a, L+1))
        (e1 e2) → (substitute(e1, a, L)
                   substitute(e2, a, L))
        ∑i ei: pi → ∑i substitute(ei, a, L): pi
```

As noted above, we have defined two additional reductions that we call γ_L and γ_R. The γ_R reduction is essential for reducing applications where the β reduction cannot be applied. Continuing the example introduced earlier in the paper:

$$(\lambda\ 1\ F\ 1)\{T: 0.6, F: 0.4\} \xrightarrow{\gamma_R} \{((\lambda\ 1\ F\ 1)\ T): 0.6, ((\lambda\ 1\ F\ 1)\ F): 0.4)\}$$

Now since both T and F do not contain distributions, β reductions can be applied:

$$\{((\lambda\ 1\ F\ 1)\ T): 0.6, ((\lambda\ 1\ F\ 1)\ F): 0.4)\} \xrightarrow{\beta} \{(T\ F\ T): 0.6, ((\lambda\ 1\ F\ 1)\ F): 0.4\}$$

$$\{(T\ F\ T): 0.6, ((\lambda\ 1\ F\ 1)\ F): 0.4\} \xrightarrow{\beta} \{(T\ F\ T): 0.6, (F\ F\ F): 0.4\}$$

And now, using the definitions of T and F it is easy to see that (T F T) and (F F F) both are reducible to F.

4. Inference

The task of inference in our stochastic λ-calculus is the same as the problem of finding a normal form for an expression. In standard λ-calculus a normal form is a term to which no β reduction can be applied. In the stochastic version, this must be modified to be any term to which no β or γ reduction can be applied. It is a relatively simple task to extend the Church-Rosser theorem [Hindley and Seldin, 1986; deBruijn, 1972] to show that this normal form, when it exists for a given expression, is unique. Thus one can construct inference algorithms in a manner similar to doing evaluation in a λ-calculus system.

4.1 A Simple Inference Algorithm

We next show a simple algorithm for doing such evaluation. This algorithm doesn't reduce to a normal form, rather to the equivalent of a weak head normal form [Reade, 1989].

```
peval(expr) = case expr
    (λ e) → expr
    (e1 e2) → papply(peval(e1), e2)
    ∑i ei: pi → ∑i peval(ei): pi

papply(f, a) = case f
    ∑i fi: pi → ∑i papply(fi, a)::pi
    (λ fe) → case a
          (λ e) → peval(substitute(f, a))
          (e1 e2) → papply(f, peval(a))
          ∑i ei: pi → ∑i papply(f, ei): pi
```

Peval and papply are the extended version of eval and apply from languages such as LISP. Peval implements left outermost first evaluation for function applications ($(e_1\ e_2)$). For λ-abstractions ,$(\lambda\ e)$, no further evaluation is needed (it would be if one wanted a true normal form). For distributions, it evaluates each term in the set and then performs a weighted sum.

Papply uses a γ_L reduction when a distribution is being applied to some operand. When a λ-abstraction is being applied, its behavior depends on the operand. When the operand is an abstraction, it applies a β reduction. If the operand is an application, it uses eager evaluation (evaluating the operand). When the operand is a distribution, it applies a γ_R reduction.

4.2 Efficiency Issues

We have presented this simple, but not optimal, algorithm for purposes of clarity. One key problem is that it uses lazy evaluation only when the operand is a λ-abstraction. One would like to use lazy evaluation as much as possible. An obvious improvement would be to check to see if the bound variable in an operator is used more than one time. If it is used only once (or not at all) then lazy evaluation can be used regardless of whether the operand will evaluate to a distribution. This is true because there are no correlations in the different instances of the bound variable to keep track of, as there is at most one such instance.

Another potential improvement would be to expand the set of cases in which it is determined that the operand cannot be reduced to a distribution. To make this determination in all cases is as hard as evaluating the operand, which is exactly what one tries to avoid through lazy evaluation. However, some cases may be easy to detect. For example, an expression that doesn't contain any distributions in its parse tree clearly will not evaluate to a distribution. Finally, we may tailor the algorithm using the reductions in different orders for particular application domains. The algorithm we presented doesn't utilize the η reduction, which may help in some cases. Also identifying more cases when β reductions can be applied may allow for more efficient algorithms for specific applications.

4.3 Caching

Efficient computational inference in probabilistic systems generally involves the saving and reuse of partial and intermediate results [Koller et al., 1997]. Algorithms for BBNs as well as for HMMs and other stochastic problems are often based on some form of dynamic programming [Dechter, 1996, Koller et al., 1997]. Using deBruijn notation makes caching expressions easy. Without the ambiguity that arises from the arbitrary choice of variable names (α-equivalence), one needs only to find exact matches for expressions.

5. Conclusions and Future Work

We have presented a formal framework for recursive modeling languages. We are currently working on extending the family of algorithms in a systematic way to include approximation schemes. It would be useful to analyze the efficiency of various algorithms on standard problems (such as polytrees [Pearl, 1988]) where the efficiency of the optimal algorithm is known. This may point to optimal reduction orderings and other improvements to inference. We are also looking at constructing formal models of the semantics of the language. Finally, we are considering the implications of moving from the pure λ-calculus presented here to an applicative λ-calculus. The results of that representational change, along with type inference mechanisms, may be important in further development in the theory of recursive stochastic modeling languages.

6. Acknowledgements

We will acknowledge financial support and useful discussions in the final paper.

References

[deBruijn, 1972] N.G. deBruijn. Lambda Calculus Notation with Nameless Dummies, A Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem. In *Indagationes mathematicae*. 34:381-392 1972.

[Dechter, 1996] R. Dechter. Bucket elimination: A unifying framework for probabilistic inference. In *Proceedings of the Twelfth Annual Conference on Uncertainty in Artificial Intelligence (UAI-96)*. 1996.

[Hindley and Seldin, 1989] J.R. Hindley and J.P. Seldin. *Introduction to Combinators and l-Calculus.* Cambridge, UK: Cambridge University Press. 1989.

[Koller et al., 1997] D. Koller, D. McAllester, and A. Pfeffer. Effective Bayesian Inference for Stochastic Programs. In *Proceedings of American Association of Artificial Intelligence Conference*, Cambridge: MIT Press. 1997.

[Koller and Pfeffer, 1997] D. Koller and A. Pfeffer. Object-oriented Bayesian Networks. In *Proceedings of the Thirteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI-97)*, San Francisco: Morgan Kaufmann. 1997.

[Koller and Pfeffer, 1998] D. Koller and A. Pfeffer. Probabilistic Frame-Based Systems. In *Proceedings of American Association of Artificial Intelligence Conference*, Cambridge: MIT Press. 1998.

[Laskey and Mahoney, 1997] K. Laskey and S. Mahoney. Network Fragments: Representing Knowledge for Constructing Probabilistic Models. In *Proceedings of the Thirteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI-97)*, San Francisco: Morgan Kaufmann. 1997.

[Pearl, 1988] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*, San Mateo CA: Morgan Kaufmann. 1988.

[Pearl, 2000] J. Pearl. *Causality*. Cambridge, UK: Cambridge University Press. 2000.

[Pfeffer, 2000] A. Pfeffer. *Probabilistic Reasoning for Complex systems*. Ph.D. Dissertation, Stanford University. 2000.

[Pfeffer and Koller, 2000] A. Pfeffer and D. Koller. Semantics and Inference for Recursive Probability Models. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence*. 538-544 Cambridge: MIT Press. 2000.

[Pfeffer et al., 1999] A. Pfeffer, D. Koller, B. Milch, and K. Takusagawa. SPOOK: A System for Probabilistic Object-Oriented Knowledge Representation. In *Proceedings of the 15th Annual Conference on Uncertainty in AI (UAI)*, San Francisco: Morgan Kaufmann. 1999.

[Pless et al., 2000] D. Pless, G. Luger, and C. Stern. A New Object-Oriented Stochastic Modeling Language. *Proceedings of the IASTED International Conference*, Zurich: IASTED/ACTA Press. 2000.

[Reade, 1989] C. Reade. *Elements of Functional Programming*. New York: Addison-Wesley. 1989.

[Xiang et al., 2000] Y. Xiang, K.G. Olesen and F.V. Jensen. Practical Issues in Modeling Large Diagnostic Systems with Multiply Sectioned Bayesian Networks, *International Journal of Pattern Recognition and Artificial Intelligence*. 2000.

[Xiang et al., 1993] Y. Xiang, D. Poole, and M. Beddoes. Multiply Sectioned Bayesian Networks and Junction Forests for Large Knowledge-Based Systems. *Computational Intelligence*, 9(2): 171-220. 1993.