Noname manuscript No.
(will be inserted by the editor)

Probabilistic Modelling, Inference and Learning using Logical Theories

K.S. Ng · J.W. Lloyd · W.T.B. Uther

Received: date / Accepted: date

Abstract This paper provides a study of probabilistic modelling, inference and learning in a logic-based setting. We show how probability densities, being functions, can be represented and reasoned with naturally and directly in higher-order logic, an expressive formalism not unlike the (informal) everyday language of mathematics. We give efficient inference algorithms and illustrate the general approach with a diverse collection of applications. Some learning issues are also considered.

1 Introduction

Complex computer applications, especially those needing the technology of artificial intelligence, demand the ability to model structured knowledge and uncertainty, and typically require the use of procedures that acquire new knowledge. The associated research issues arising from dealing with such applications are thus at the intersection of several subfields, including logic, probability theory, and machine learning. Given the increasing importance of such complex applications, it is not surprising that there is a huge body of literature mostly from researchers in the artificial intelligence community that addresses the myriad associated research issues. Useful surveys of the more pertinent papers in this literature can be found in [9], [41], and [10].

The nature of much of this literature is partly explained by the fact that the logical and the probabilistic artificial intelligence communities have historically been separated and have only started to come together in the last decade or two. A consequence of this separation is that most of the probabilistic reasoning techniques were originally developed for the propositional case; thus researchers have had to spend much of the

K.S. Ng
National ICT Australia, The Australian National University
E-mail: keesiong.ng@nicta.com.au

J.W. Lloyd
College of Engineering and Computer Science, The Australian National University
E-mail: john.lloyd@anu.edu.au

W.T.B. Uther
National ICT Australia, University of New South Wales
E-mail: william.uther@nicta.com.au

last decade lifting them to the first-order case. Here we reconsider the entire issue from the perspective of a general and well-established principle.

The fundamental principle on which we rely is that of the axiomatic method: given some situation that one wants to capture, one writes down a logical theory that has the intended interpretation as a model; then one can determine the value of any term in the intended interpretation by a (sound) reasoning procedure. This is a long held principle that has driven the development of logic for over a century, but has often been considered unsuitable for modelling uncertainty because of the perceived limitation that a (classical) logic can only capture the truth or falsity of a formula, but nothing 'in between'. We demonstrate in this paper that the axiomatic method in partnership with a suitable logic is more than powerful enough to satisfactorily model uncertainty.

The crucial issue is exactly how the uncertainty (which we model with probability theory) is captured. Almost all other approaches to this issue have a clear separation between the logical statements and the probabilities. In contrast, we prefer *the uncertainty to be captured directly in the theory itself*. Taking this for granted, the issue becomes one of finding a suitable logic. There has been a tradition of extending first-order logic with probabilistic facilities. One could follow this approach, but here we follow a different one and use a logic that was shown over half a century ago to have exactly the desired properties: higher-order logic [6, 27, 2].

The best way to think about higher-order logic is that it is the formalisation of everyday informal mathematics: whatever mathematical description one might give of some situation, the formalisation of that situation in higher-order logic is likely to be a straightforward translation of the informal description. In particular, higher-order logic provides a suitable foundation for mathematics itself which has many advantages over more traditional approaches that are based on axiomatising sets in first-order logic [2]. The most crucial property of higher-order logic that we exploit is that it admits so-called higher-order functions which take functions as arguments and/or return functions as results. It is this property that allows the modelling of, and reasoning about, probabilistic concepts directly in higher-order theories, and thus provides an elegant solution to the problem of integrating logic and probability [43].

Let us examine this idea in a little more detail. Applications are typically modelled mathematically by functions (this includes predicates and constants), so that suitable theories consist mainly of function definitions. Therefore the most useful and direct approach to handling uncertainty is to model the uncertainty in (some) function definitions in theories. Consider a function $f : \sigma \to \tau$ for which there is some uncertainty about its values that we want to model. We do this with a function

$$f' : \sigma \to \mathit{Density}\ \tau,$$

where, for each argument t, the value of the function f' applied to t, written $(f'\ t)$, is a suitable probability density for modelling the uncertainty in the value of $(f\ t)$. (Here $\mathit{Density}\ \tau$ is the type of densities on the domain of type τ.) The intuition is that the actual value of $(f\ t)$ is likely to be where the 'mass' of the density $(f'\ t)$ is most concentrated. Of course, (unconditional) densities can also be expressed by functions having a signature of the form $\mathit{Density}\ \tau$. This simple idea turns out to be a powerful and convenient way of modelling uncertainty with logical theories in diverse applications. Note carefully the use that has been made of higher-order logic here to model functions whose values are densities.

There are a number of requirements that follow from our decision to use higher-order logic. The first is that we need a reasoning system for the logic in which to

carry out probabilistic computations. This is provided by the Bach system which is a functional logic programming language [38]. Bach programs are equational theories in higher-order logic and these are what we employ to model applications. Thus probabilistic reasoning is handled by Bach's computation system which is an equational reasoning system with special support to make probabilistic reasoning efficient. This is provided by equations that implement (the equivalent of) variable elimination and lifted inference. We also need to show that we can easily model suitable probabilistic concepts and provide operations on them in the logic. We argue that all that is really needed are probability densities and these can be easily modelled and reasoned about in both the discrete and continuous cases in higher-order logic. Finally, we need to demonstrate the usefulness and practicality of the ideas and this we do using a diverse set of applications, most of which have also been considered by other authors and thus provide a comparison with alternative approaches.

We have taken the view that we want Bach to be a general-purpose reasoning system. Therefore we have designed it to handle all the various kinds of applications that other systems can. The price we pay for this generality is that for a particular application a more specialised probabilistic reasoning system may have some advantages. With extra work, Bach could also be fined-tuned to be just as effective as a more specialised system, but we leave this for subsequent development.

The paper is organised as follows. Section 2 gives an outline of the underlying logic and recalls some pertinent probabilistic concepts. Section 3 presents our general approach to probabilistic modelling. Section 4 describes the reasoning system. Section 5 considers some associated learning problems. Section 6 contains a general discussion of our approach in the context of related work.

2 Mathematical Preliminaries

We review a standard formulation of higher-order logic based on Church's simple theory of types [6] in Section 2.1. (More complete accounts of the logic can be found in [34] and [35]. Other references on higher-order logic include [57], [2], [56], [32], and [54].) We then introduce a modicum of measure theory in Section 2.2 and show how probability density functions can be naturally represented in the logic.

2.1 Logic

Definition 1 An *alphabet* consists of three sets:

1. a set $\mathfrak{T}$ of type constructors;
2. a set $\mathfrak{C}$ of constants; and
3. a set $\mathfrak{V}$ of variables.

Each type constructor in $\mathfrak{T}$ has an arity, which is the number of arguments it needs. The set $\mathfrak{T}$ always includes the type constructor Ω of arity 0. Ω is the type of the booleans. Each constant in $\mathfrak{C}$ has a type signature. The set $\mathfrak{V}$ is denumerable. Variables are typically denoted by $x, y, z, \ldots$.

Types are built up from the set of type constructors, using the symbols $\rightarrow$ and $\times$.

Definition 2 A *type* is defined inductively as follows.

1. If T is a type constructor of arity k and $\alpha_1, \ldots, \alpha_k$ are types, then $T\ \alpha_1 \ldots \alpha_k$ is a type. (Thus a type constructor of arity 0 is a type.)
2. If α and β are types, then $\alpha \to \beta$ is a type.
3. If $\alpha_1, \ldots, \alpha_n$ are types, then $\alpha_1 \times \cdots \times \alpha_n$ is a type.

We use the convention that $\to$ is right associative. So, for example, when we write $\alpha \to \beta \to \gamma \to \kappa$ we mean $\alpha \to (\beta \to (\gamma \to \kappa))$.

Besides Ω, here are some other common types we will need.

Example 1 The type of the integers is denoted by *Int*, and the type of the reals by *Real*. Also $(List\ \sigma)$ is the type of lists whose items have type σ, and $\{\sigma\}$ is the type of sets whose elements have type σ. The type $\{\sigma\}$ is a synonym for $\sigma \to \Omega$.

The set $\mathfrak{C}$ always includes the following constants: $\top$ and $\bot$ having signature Ω; $=_\alpha$ having signature $\alpha \to \alpha \to \Omega$ for each type α; $\neg$ having signature $\Omega \to \Omega$; $\wedge$, $\vee$, and $\longrightarrow$ having signature $\Omega \to \Omega \to \Omega$; and Σ_α and Π_α having signature $(\alpha \to \Omega) \to \Omega$ for each type α. The intended meaning of $\top$ is true, and that of $\bot$ is false. The intended meaning of $=_\alpha$ is identity (that is, $(=_\alpha\ x\ y)$ is $\top$ iff x and y are identical), and the intended meanings of the connectives $\neg$, $\wedge$, $\vee$, and $\longrightarrow$ are as usual (*not*, *and*, *or* and *implies*). The intended meaning of Σ_α is that Σ_α maps a predicate (i.e. boolean function) over elements of type α to $\top$ iff the predicate maps at least one element of type α to $\top$. The intended meaning of Π_α is that Π_α maps a predicate over elements of type α to $\top$ iff the predicate maps all elements of type α to $\top$.

Other useful constants that often appear in applications include the integers, real numbers, and data constructors like $\#_\sigma : \sigma \to (List\ \sigma) \to (List\ \sigma)$ and $[]_\sigma : List\ \sigma$ for constructing lists where the elements have type σ. (See Example 3.) The notation $C : \sigma$ is used to denote that the constant C has signature σ.

Definition 3 A *term*, together with its type, is defined inductively as follows.

1. A variable in $\mathfrak{V}$ of type α is a term of type α.
2. A constant in $\mathfrak{C}$ having signature α is a term of type α.
3. If t is a term of type β and x a variable of type α, then $\lambda x.t$ is a term of type $\alpha \to \beta$.
4. If s is a term of type $\alpha \to \beta$ and t a term of type α, then $(s\ t)$ is a term of type β.
5. If $t_1, \ldots, t_n$ are terms of type $\alpha_1, \ldots, \alpha_n$, respectively, then $(t_1, \ldots, t_n)$ is a term of type $\alpha_1 \times \cdots \times \alpha_n$.

Terms of the form $(\Sigma_\alpha\ \lambda x.t)$ are written as $\exists_\alpha x.t$ and terms of the form $(\Pi_\alpha\ \lambda x.t)$ are written as $\forall_\alpha x.t$ (in accord with the intended meaning of Σ_α and Π_α). Thus, in higher-order logic, each quantifier is obtained as a combination of an abstraction acted on by a suitable function (Σ_α or Π_α). A *formula* is a term of type Ω. The universal closure of a formula φ is denoted by $\forall(\varphi)$.

Example 2 Constants like $\top$, 42, 3.11, and $+$ (a boolean, an integer, a real number and a function with signature $Int \to Int \to Int$) are terms. Variables like x, y, z are terms. An example of a term that can be formed using abstraction is $\lambda x.((+\ x)\ x)$, whose intended meaning is a function that takes a number x and returns $x + x$. To apply that function to the constant 42, for example, we use application to form the term $(\lambda x.((+\ x)\ x)\ 42)$.

Example 3 The term $(\#_{Int}\ 2\ (\#_{Int}\ 3\ []_{Int}))$ represents a list with the numbers 2 and 3 in it, obtained via a series of applications from the constants $\#_{Int}, []_{Int}, 2$, and 3, each of which is a term. For convenience, we sometimes write $[2, 3]$ to represent the same list.

Example 4 Sets are identified with predicates in the logic. In other words, sets are represented by their membership functions. Thus, the term

$$\lambda x.((\vee\ ((=_{Int}\ x)\ 2))\ ((=_{Int}\ x)\ 3)) \tag{1}$$

can be used to represent a set with the integers 2 and 3 in it. We often use infix notation for common function symbols like equality and the connectives. We also adopt the convention that applications are left-associative; thus, $(f\ x\ y)$ means $((f\ x)\ y)$. All these conventions allow us to write the more natural $\lambda x.((x =_{Int} 2) \vee (x =_{Int} 3))$ for (1) above. For convenience, we sometimes also write $\{2, 3\}$ to represent the same set. Since sets are predicates, set membership test is obtained using function application. Let s denote (1) above. To check whether a number y is in s, we write $(s\ y)$.

The polymorphic version of the logic extends what is given above by also having available type variables (denoted by $a, b, c, \ldots$). The definition of a type as above is then extended to polymorphic types that may contain type variables and the definition of a term as above is extended to terms that may have polymorphic types. We work in the polymorphic version of the logic in the remainder of the paper. In this case, we drop the α in constants like $\exists_\alpha$, $\forall_\alpha$, $=_\alpha$, $\#_\alpha$, and $[]_\alpha$, since the types associated with these are now inferred from the context.

Example 5 A common polymorphic function we need is $\mathit{if\_then\_else} : \Omega \times a \times a \to a$. Using it, we can give the following equivalent way of representing the set denoted by (1) above:

$$\lambda x.(\mathit{if\_then\_else}\ ((x = 2), \top, (\mathit{if\_then\_else}\ ((x = 3), \top, \bot)))).$$

Using *if* x *then* y *else* z as syntactic sugar for $(\mathit{if\_then\_else}\ (x, y, z))$, the above can be written in the following more readable form:

$$\lambda x.\mathit{if}\ x = 2\ \mathit{then}\ \top\ \mathit{else}\ \mathit{if}\ x = 3\ \mathit{then}\ \top\ \mathit{else}\ \bot.$$

The logic can be given a rather conventional Henkin semantics – more on this later.

2.2 Densities

Probability distributions can be described formally using measure theory. A probability distribution is defined over some set X and observable events correspond to subsets of X. The collection $\mathcal{A}$ of these subsets is called the measurable sets of X and we require that $\mathcal{A}$ forms a σ-algebra; in other words, $\mathcal{A}$ satisfies the following two properties: (i) $X \in \mathcal{A}$; and (ii) $\mathcal{A}$ is closed under complementation and countable unions. A measure is a function μ that maps each measurable set in $\mathcal{A}$ to a real number in the range $[0, \infty]$ and that is countably additive. A triple $(X, \mathcal{A}, \mu)$ satisfying the above is called a measure space. Let Y be another set and $\mathcal{B}$ a σ-algebra on Y. A function $f : X \to Y$ is measurable if $f^{-1}(B) \in \mathcal{A}$, for all $B \in \mathcal{B}$.

Definition 4 Let $(X, \mathcal{A}, \mu)$ be a measure space and $f : X \to \mathbb{R}$ a measurable function. Then f is a *density (wrt the measure μ)* if (i) $f(x) \geq 0$, $\forall x \in X$, and (ii) $\int_X f \, d\mu = 1$.

There are two main cases of interest. The first is when μ is the counting measure on X, in which case $\int_X f \, d\mu = \sum_{x \in X} f(x)$; this is the discrete case. The second case is when X is $\mathbb{R}^n$, for some $n \geq 1$, and μ is the Lebesgue measure; this is the continuous case.

To formulate the above ideas in the logic, we introduce a type synonym

$$\mathit{Density}\ a \equiv a \to \mathit{Real}.$$

Here a is a type variable; so *Density* a is a polymorphic type. The intended meaning of a term of type *Density* τ in the semantics is a density over $\mathcal{D}_\tau$, the domain of τ, and not some arbitrary real-valued function. Any term of type *Density* τ, for some τ, is called a *density*.

In performing probabilistic inference, we often need to compose densities and functions in different ways. Two composition functions we will need in this paper are given next. The definitions are *mathematical* definitions; we will see how they are defined *in the logic* in Section 4.2.1.

Definition 5 The function

$$\S : \mathit{Density}\ Y \to (Y \to \mathit{Density}\ Z) \to \mathit{Density}\ Z$$

is defined by

$$(f \S g)(z) = \int_Y f(y) \times g(y)(z) \; d\nu(y),$$

where $f : \mathit{Density}\ Y$, $g : Y \to \mathit{Density}\ Z$, and $z \in Z$.

Specialised to the discrete case, the definition is

$$(f \S g)(z) = \sum_{y \in Y} f(y) \times g(y)(z).$$

Intuitively, § is used to chain together a density on Y and a conditional density on Z to produce a density on Z by marginalising out Y.

Definition 6 The function

$$\$: \mathit{Density}\ Y \to (Y \to Z) \to \mathit{Density}\ Z$$

is defined by

$$(f \$ g)(z) = \int_{g^{-1}(z)} f(y) \; d\nu(y),$$

where $f : \mathit{Density}\ Y$, $g : Y \to Z$, and $z \in Z$.

Specialised to the discrete case, the definition is

$$(f \$ g)(z) = \sum_{y \in g^{-1}(z)} f(y).$$

Intuitively, \$ is used to obtain a density on Z from a density on Y by passing it through a function from Y to Z.

3 Modelling

We will use three examples to illustrate the central concepts and also the generality of our approach to probabilistic modelling. Example 6 illustrates the kind of problem being investigated with formalisms like stochastic logic programs [42]. Example 8 presents a solution to a generative probabilistic model problem described in [40]. We will focus only on modelling issues in this section, delaying computational issues to Section 4.

Example 6 Consider an agent that makes recommendations of TV programs to a user. The agent has access to the TV guide through the definition of the function *tv_guide*. It also knows about the user's preferences for TV programs through the definition of the function *likes*, the uncertainty of which is modelled by densities in its codomain. Suppose now that the agent is asked to make a recommendation about a program at a particular occurrence (that is, date, time, and channel), except that there is some uncertainty in the actual occurrence intended by the user; this uncertainty, which is modelled in the definition of the density *choice*, could come about, for example, if the user asked the somewhat ambiguous question "Are there any good programs on ABC during dinner time?". The question the agent needs to answer is the following: given the uncertainty in *choice* and *likes*, what is the probability that the user likes the program that is on at the occurrence intended by the user.

This situation is modelled as follows. First, here are some type synonyms.

$$Occurrence = Date \times Time \times Channel$$
$$Program = Title \times Duration \times Genre \times Classification \times Synopsis.$$

Here is the definition of the density *choice* that models the uncertainty in the intended occurrence.

$$\begin{aligned} &choice : Density\ Occurrence \\ &\forall x.((choice\ x) = \mathit{if}\ x = ((21,7,2007),(19,30),ABC)\ \mathit{then}\ 0.8 \\ &\qquad\qquad \mathit{else\ if}\ x = ((21,7,2007),(20,30),ABC)\ \mathit{then}\ 0.2\ \mathit{else}\ 0). \end{aligned} \tag{2}$$

So the uncertainty is in the time of the program; it is probably 7.30pm, but it could be 8.30pm.

Next there is the TV guide that maps occurrences to programs.

$$\begin{aligned} &tv\_guide : Occurrence \rightarrow Program \\ &\forall x.((tv\_guide\ x) = \\ &\quad \mathit{if}\ (x = ((20,7,2007),(11,30),Win)) \\ &\qquad\qquad \mathit{then}\ (\text{“}NFL\ Football\text{”}, 60, Sport, G, \text{“}The\ Buffalo\ \ldots\text{”}) \\ &\quad \mathit{else\ if}\ (x = ((21,7,2007),(19,30),ABC)) \\ &\qquad\qquad \mathit{then}\ (\text{“}Seinfeld\text{”}, 30, Sitcom, PG, \text{“}Kramer\ \ldots\text{”}) \\ &\quad \mathit{else\ if}\ (x = ((21,7,2007),(20,30),ABC)) \\ &\qquad\qquad \mathit{then}\ (\text{“}The\ Bill\text{”}, 50, Drama, M, \text{“}Sun\ Hill\ \ldots\text{”}) \\ &\qquad \vdots \\ &\quad \mathit{else}\ (\text{“ ”}, 0, NULL, NA, \text{“ ”})). \end{aligned} \tag{3}$$

Finally, here is the definition of the function *likes*. We use $\langle(\top, r_1), (\bot, r_2)\rangle$ as a shorthand for the term

$\lambda y.\mathit{if}\ y = \top\ \mathit{then}\ r_1\ \mathit{else\ if}\ y = \bot\ \mathit{then}\ r_2\ \mathit{else}\ 0$

in the following.

$$
\begin{aligned}
&\mathit{likes} : \mathit{Program} \to \mathit{Density}\ \Omega\\
&\forall x.((\mathit{likes}\ x) =\\
&\qquad \mathit{if}\ (\mathit{projTitle}\ x) = \text{“}\mathit{NFL\ Football}\text{”}\ \mathit{then}\ \langle(\top, 1), (\bot, 0)\rangle\\
&\qquad \mathit{else\ if}\ (\mathit{projGenre}\ x) = \mathit{Movie}\ \mathit{then}\ \langle(\top, 0.75), (\bot, 0.25)\rangle\\
&\qquad \mathit{else\ if}\ (\mathit{projGenre}\ x) = \mathit{Documentary}\ \mathit{then}\ \langle(\top, 1), (\bot, 0)\rangle\\
&\qquad \mathit{else\ if}\ (\mathit{projTitle}\ x) = \text{“}\mathit{World\ Soccer}\text{”}\ \mathit{then}\ \langle(\top, 0.9), (\bot, 0.1)\rangle\\
&\qquad \mathit{else\ if}\ (\mathit{projGenre}\ x) = \mathit{Drama}\ \mathit{then}\ \langle(\top, 0.2), (\bot, 0.8)\rangle\\
&\qquad \mathit{else\ if}\ (\mathit{projGenre}\ x) = \mathit{Sitcom}\ \mathit{then}\ \langle(\top, 0.9), (\bot, 0.1)\rangle\\
&\qquad \mathit{else}\ \langle(\top, 0), (\bot, 1)\rangle).
\end{aligned}
\tag{4}
$$

The definition of *likes* given above can come about from a learning process like [8].

Recall that the agent needs to compute the probability that the user likes the program which is on at the occurrence intended by the user. This amounts to computing the value of the term

$$((\mathit{choice}\ \$\ \mathit{tv\_guide})\ \S\ \mathit{likes}), \tag{5}$$

which is a term of type *Density* Ω. We show in Section 4.3.1 how (5) simplifies to

$\lambda z.\mathit{if}\ z = \top\ \mathit{then}\ 0.76\ \mathit{else\ if}\ z = \bot\ \mathit{then}\ 0.24\ \mathit{else}\ 0.$

The agent can now use this information to decide whether to recommend the program or not.

Example 7 The probabilistic model presented in this example is motivated by work on lifted inference [50, 10]. Consider the factor graph [30] representation of an undirected graphical model shown in Fig. 1, where each node is a boolean random variable and the two common factors f and g are defined as follows:

$$
\begin{aligned}
&f : \Omega \to \Omega \to \mathit{Real}\\
&(f\ \top\ \top) = 7.0 \qquad (f\ \bot\ \top) = 0.5\\
&(f\ \top\ \bot) = 3.0 \qquad (f\ \bot\ \bot) = 9.5
\end{aligned}
$$

$$
\begin{aligned}
&g : \Omega \to \Omega \to \mathit{Real}\\
&(g\ \top\ \top) = 1.0 \qquad (g\ \bot\ \top) = 0.5\\
&(g\ \top\ \bot) = 9.0 \qquad (g\ \bot\ \bot) = 9.5.
\end{aligned}
$$

We can represent the factor graph in a propositional way directly in Bach but the size of the resultant theory would be unnecessarily large. In the following, we exploit the repeated structure in the graph to arrive at a more compact representation. As we shall see in Section 4.3.2, this compact representation can also lead to much better inference procedures.

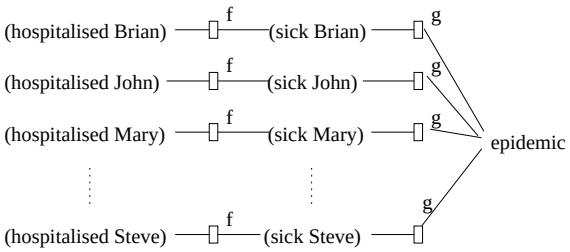


Fig. 1 A factor graph with repeated structures

We first note that all the random variables except *epidemic* can be grouped under two classes: one parameterised by the function $hospitalised : Person \to \Omega$, and the other by the function $sick : Person \to \Omega$. Observe now that there is a one-to-one correspondence between an assignment of values to the variables parameterised by *hospitalised* and a definition for the function *hospitalised*. Similarly for the variables parameterised by *sick*. This leads us to define the underlying joint density for the factor graph by defining the following real-valued function over triples (s, h, e), where s (short for *sick*) and h (short for *hospitalised*) are function variables and e (short for *epidemic*) is a boolean variable.

$$joint : (Person \to \Omega) \times (Person \to \Omega) \times \Omega \to Real$$
$$(joint\ (s, h, e)) = \prod_{x \in all} (f\ (s\ x)\ (h\ x)) \times (g\ (s\ x)\ e)$$
$$all : Person \to \Omega$$
$$all = \{Brian, John, Mary, Mike, Liz, Steve\}.$$

Here, $\prod_{x \in all}$ is used to form a product of terms over the set *all*. The Bach definition of $\prod$ is given later in Section 4.2.1.[1]

One can answer various queries using the model. For example, suppose we know Brian, John and Mary are hospitalised but Mike is not and we would like to compute the probability of there being an epidemic. In other words, we want to compute

$$Pr(e = \top \mid (h\ Brian) = \top, (h\ John) = \top, (h\ Mary) = \top, (h\ Mike) = \bot). \tag{6}$$

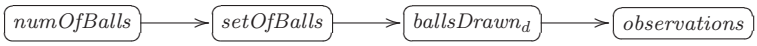
This can be solved by marginalising out s and those parts of h not already instantiated in the evidence from *joint*. This involves summations over sets of functions and we will see how that can be achieved in Section 4.3.2.

Example 8 We model the following scenario, which is one of the main examples used in [40]. An urn contains an unknown number of balls that have the colour blue or green with equal probability. Identically coloured balls are indistinguishable. An agent has the prior belief that the distribution of the number of balls is a Poisson distribution with mean 6. The agent now draws some balls from the urn, observes their colour, and then replaces them. The observed colour is different from the actual colour of the ball

[1] The function $\prod$ is distinct from the function Π used to obtain universal quantification in Section 2.1.

drawn with probability 0.2. On the basis of these observations, the agent should infer certain properties about the urn, such as the number of balls it contains.

This is an interesting problem because it is an example of a model that exhibits domain uncertainty: the number of balls in the urn is unknown and *unbounded*. This, claims [40], is the reason it cannot be modelled using some existing first-order probabilistic languages. Interestingly, the problem can be modelled rather straightforwardly if we can define densities over structured objects like sets and lists. There are essentially four variables in the problem setup: the number n of balls in the urn, the actual set s of balls in the urn, the list x of balls drawn, and the list y of observations made. The following is a suitable graphical model.



The details of the graphical model are specified in the following theory.

$$colour : Colour \to \Omega$$
$$(colour\ x) = (x = Blue) \vee (x = Green)$$

$$numOfBalls : Density\ Int$$
$$(numOfBalls\ x) = (poisson\ 6\ x)$$

$$poisson : Int \to Density\ Int$$
$$(poisson\ x\ y) = e^{-x}x^y/y!$$

$$setOfBalls : Int \to Density\ \{Ball\}$$
$$(setOfBalls\ n\ s) =$$
$$\quad if\ \exists x_1 \cdots \exists x_n.((colour\ x_1) \wedge \cdots \wedge (colour\ x_n) \wedge (s = \{(1, x_1), \ldots, (n, x_n)\}))$$
$$\quad then\ 0.5^n\ else\ 0$$

$$ballsDrawn : Int \to \{Ball\} \to Density\ (List\ Ball)$$
$$(ballsDrawn\ d\ s\ x) =$$
$$\quad if\ \exists x_1 \cdots \exists x_d.((s\ x_1) \wedge \cdots \wedge (s\ x_d) \wedge (x = [x_1, \ldots, x_d]))\ then\ (card\ s)^{-d}\ else\ 0$$

$$observations : (List\ Ball) \to Density\ (List\ Colour)$$
$$(observations\ x\ y) = if\ (length\ x) = (length\ y)\ then\ (obsProb\ x\ y)\ else\ 0$$

$$obsProb : (List\ Ball) \to (List\ Colour) \to Real$$
$$(obsProb\ []\ []) = 1$$
$$(obsProb\ (\#\ (v, y_1)\ z_1)\ (\#\ y_2\ z_2)) =$$
$$\qquad (if\ (y_1 = y_2)\ then\ 0.8\ else\ 0.2) \times (obsProb\ z_1\ z_2)$$

$$joint : Int \to Density\ (Int \times \{Ball\} \times (List\ Ball) \times (List\ Colour))$$
$$(joint\ d\ (n, s, x, y)) =$$
$$\quad (numOfBalls\ n) \times (setOfBalls\ n\ s) \times (ballsDrawn\ d\ s\ x) \times (observations\ x\ y)$$

The function *card* returns the cardinality of a set; *length* returns the size of a list. The functions *setOfBalls* and *ballsDrawn* are defined informally above; formal recursive definitions can be given.

The function *joint* at the end defines the following mathematical expression

$$Pr(n)Pr(s \mid n)Pr(x \mid s)Pr(y \mid x).$$

The prior over the number of balls is captured in the density *numOfBalls*. Given n the number of balls in the urn, (*setOfBalls* n) specifies a distribution over all possible sets of balls in the urn. A ball is represented by an integer identifier and its colour: $Ball = Int \times Colour$. Every set of balls that has non-zero weight under (*setOfBalls* n) contains n balls. These balls are labelled 1 to n and their colours are chosen randomly. Given d the number of draws the agent can make and s the set of balls in the urn, (*ballsDrawn* d s) specifies a distribution over all possible sequences of balls drawn, where each draw sequence is represented as a list of balls. Finally, given x the list of balls drawn, (*observations* x) gives the distribution over the possible colours that is actually observed by the agent.

We can compute marginalisations of the given joint distribution to obtain answers to different questions, questions like "How many balls are in the urn?" and "Was the same ball drawn twice?". For example, the following gives the probability that the number of balls in the urn is m after we have seen the colours $[o_1, o_2, \ldots, o_d]$ from d draws:

$$\frac{1}{K} \sum_s \sum_l (joint\ d\ (m, s, l, [o_1, o_2, \ldots, o_d])), \tag{7}$$

where K is a normalisation constant, s ranges over sets of balls and l ranges over lists of balls. Fig. 2 shows the posterior distribution of m after drawing ten and fifteen balls and observing that they are all blue. (We will see how the probabilities are computed in the next section.) Consistent with intuition, the lower numbers of m become increasingly probable (compared to the prior Poisson distribution) as more blue balls are observed.

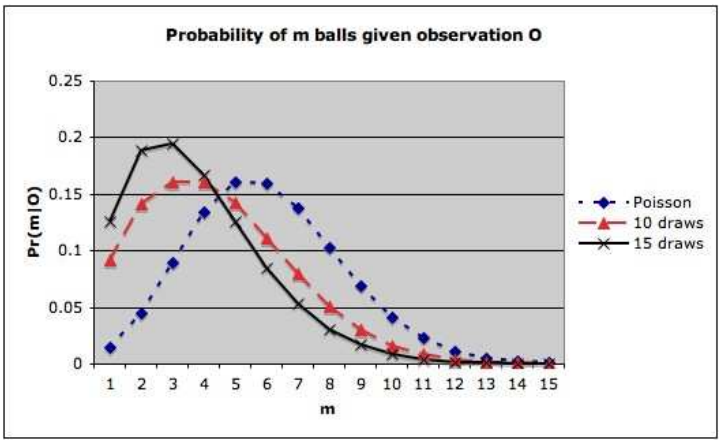


Fig. 2 The posterior distributions of the number of balls in the urn after some observations. The Poisson curve is the prior distribution on the number of balls.

4 Inference

We will first describe a general mechanism for reasoning with equational theories in Section 4.1. Extensions needed to support probabilistic reasoning are then discussed in Section 4.2. Finally, we show in Section 4.3 how the inference mechanism is used to solve the three example problems described in Section 3

4.1 Equational Reasoning

The reasoning system described next is a subset of a more general system called Bach [36,38]. It is a computation system that significantly extends existing functional programming languages by adding logic programming facilities. For convenience, we will refer to this subsystem as Bach in this paper. Bach is closely related to the Haskell programming language, being a strict superset. Haskell allows pattern matching only on data constructors. Bach extends this by also allowing pattern matching on function symbols and lambda abstractions. Further, Bach allows reduction of terms inside lambda abstractions, an operation not permitted in Haskell. It is also worth noting that any (pure) Prolog program can be mechanically translated into Bach using Clark's completion [7].

The inference mechanism underlying Bach is given in Definition 7 below. We first establish some terminology. The *occurrence* of a subterm s of t is a description of the path from the root of t to s. An occurrence of a variable x in a term is *bound* if it occurs within a subterm of the form $\lambda x.t$. Otherwise it is *free*. The notation $t[s/r]_o$ denotes the term obtained from t by replacing s at occurrence o with r. If x is a variable, the notation $t\{x/r\}$ denotes the term obtained from t by replacing every free occurrence of variable x in t with r. Two terms are α-equivalent if they are identical up to a change of bound variable names.

Definition 7 Let $\mathcal{T}$ be a theory. A *computation with respect to* $\mathcal{T}$ is a sequence $\{t_i\}_{i=1}^{n}$ of terms such that for $i = 1, \ldots, n-1$, there is a subterm s_i of t_i at occurrence o_i, a formula $\forall(u_i = v_i)$ in $\mathcal{T}$, and a substitution θ_i such that $u_i\theta_i$ is α-equivalent to s_i and t_{i+1} is $t_i[s_i/v_i\theta_i]_{o_i}$.

The term t_1 is called the *goal* of the computation and t_n is called the *answer*. Each subterm s_i is called a *redex* (short for reducible expression). The formula $\forall(t_1 = t_n)$ is called the *result* of the computation.

A selection rule chooses the redex at each step of a computation. The selection rule we use in this paper is the *leftmost* one which chooses the leftmost outermost subterm that satisfies the requirements of Definition 7. This rule gives us lazy evaluation.

We will see many examples of computations shortly. Theorem 1 below shows that computation as defined in Definition 7 is essentially a form of theorem proving. The proof of Theorem 1 is rather involved and is omitted here; the proof of a more general case of the theorem can be found in [36, Sect. 6].

Theorem 1 *The result of a computation with respect to a theory* $\mathcal{T}$ *is a logical consequence of* $\mathcal{T}$.

Computations generally require use of definitions of $=$, the connectives and quantifiers, and some other basic functions. These definitions constitute what we call the

standard equality theory. The complete list of equations in its various versions can be found in [33,35,36]. Here are some examples of equations that will be needed.

$$(\mathit{if}\ \top\ \mathit{then}\ u\ \mathit{else}\ v) = u \tag{8}$$

$$(\mathit{if}\ \bot\ \mathit{then}\ u\ \mathit{else}\ v) = v \tag{9}$$

$$(\boldsymbol{w}\ (\mathit{if}\ \boldsymbol{x} = \boldsymbol{t}\ \mathit{then}\ u\ \mathit{else}\ v)) = (\mathit{if}\ \boldsymbol{x} = \boldsymbol{t}\ \mathit{then}\ (\boldsymbol{w}\{\boldsymbol{x}/\boldsymbol{t}\}\ u)\ \mathit{else}\ (\boldsymbol{w}\ v)) \tag{10}$$

-- where $\boldsymbol{x}$ is a variable.

$$((\mathit{if}\ \boldsymbol{x} = \boldsymbol{t}\ \mathit{then}\ u\ \mathit{else}\ v)\ \boldsymbol{w}) = (\mathit{if}\ \boldsymbol{x} = \boldsymbol{t}\ \mathit{then}\ (u\ \boldsymbol{w}\{\boldsymbol{x}/\boldsymbol{t}\})\ \mathit{else}\ (v\ \boldsymbol{w})) \tag{11}$$

-- where $\boldsymbol{x}$ is a variable.

$$(x \vee y) \wedge z = (x \wedge z) \vee (y \wedge z) \tag{12}$$

$$(\lambda x.\boldsymbol{u}\ t) = \boldsymbol{u}\{x/t\} \tag{13}$$

$$\exists x_1.\cdots\exists x_n.(\boldsymbol{x} \wedge (x_1 = \boldsymbol{u}) \wedge \boldsymbol{y}) = \exists x_2.\cdots\exists x_n.(\boldsymbol{x}\{x_1/\boldsymbol{u}\} \wedge \boldsymbol{y}\{x_1/\boldsymbol{u}\}) \tag{14}$$

-- where x_1 is not free in $\boldsymbol{u}$.

$$\exists x_1.\cdots\exists x_n.(\mathbf{u} \vee \mathbf{v}) = (\exists x_1.\cdots\exists x_n.\mathbf{u}) \vee (\exists x_1.\cdots\exists x_n.\mathbf{v}) \tag{15}$$

$$\forall x_1.\cdots\forall x_n.(\boldsymbol{x} \wedge (x_1 = \boldsymbol{u}) \wedge \boldsymbol{y} \longrightarrow \boldsymbol{v}) = \forall x_2.\cdots\forall x_n.(\boldsymbol{x}\{x_1/\boldsymbol{u}\} \wedge \boldsymbol{y}\{x_1/\boldsymbol{u}\} \longrightarrow \boldsymbol{v}\{x_1/\boldsymbol{u}\}) \tag{16}$$

-- where x_1 is not free in $\boldsymbol{u}$.

Some of the equations above are schemas. A schema is intended to stand for the collection of formulas that can be obtained from the schema by replacing its syntactical variables (typeset in bold above) with terms that satisfy the side conditions, if there are any. Equations (8)-(11) above are useful for simplifying *if_then_else* expressions. Equation (13) provides β-reduction. Equations like (14)-(16) are used to provide logic programming idioms in the context of functional computations.

We now look at some example computations.

Example 9 Consider the following definition of $f : \mathit{Alphabet} \rightarrow \mathit{Int}$:

$$(f\ x) = \mathit{if}\ x = A\ \mathit{then}\ 42\ \mathit{else\ if}\ x = B\ \mathit{then}\ 21\ \mathit{else\ if}\ x = C\ \mathit{then}\ 42\ \mathit{else}\ 0. \tag{17}$$

With such a definition, it is straightforward to compute in the 'forward' direction. Thus, for example, $(f\ B)$ can be computed in the obvious way to produce the answer 21. Less obviously, the definition can be used to compute in the 'reverse' direction. For example, Fig. 3 shows the computation of the (intensional) set $\lambda x.((f\ x) = 42)$, which produces the equivalent of $\{A, C\}$ as the answer.

This next example gives further demonstration of the inbuilt capability of Bach to enumerate the elements of an intensionally described set using its logic programming facilities. Such operations play an important role in probabilistic inference routines.

Example 10 Consider the definition of *setOfBalls* in Example 8. Fig. 4 shows the computation of the term $(\mathit{setOfBalls}\ 2\ s)$. A similar computation would allow us to compute the support $\lambda s.((\mathit{setOfBalls}\ 2\ s) > 0)$ of the density $(\mathit{setOfBalls}\ 2)$, which produces the answer

$$\lambda s.(s = \{(1, \mathit{Blue}), (2, \mathit{Blue})\} \vee s = \{(1, \mathit{Blue}), (2, \mathit{Green})\} \vee s = \{(1, \mathit{Green}), (2, \mathit{Blue})\} \vee s = \{(1, \mathit{Green}), (2, \mathit{Green})\}).$$

$\lambda x.(= \underline{(f\ x)}\ 42)$ [17]

$\lambda x.\underline{(=\ (\mathit{if}\ x = A\ \mathit{then}\ 42\ \mathit{else}\ \mathit{if}\ x = B\ \mathit{then}\ 21\ \mathit{else}\ \mathit{if}\ x = C\ \mathit{then}\ 42\ \mathit{else}\ 0)}\ 42)$ [10]

$\lambda x.\underline{((\mathit{if}\ x = A\ \mathit{then}\ (=\ 42)\ \mathit{else}\ (=\ (\mathit{if}\ x = B\ \mathit{then}\ 21\ \mathit{else}\ \mathit{if}\ x = C\ \mathit{then}\ 42\ \mathit{else}\ 0)))\ 42)}$ [11]

$\lambda x.\mathit{if}\ x = A\ \mathit{then}\ \underline{(42 = 42)}\ \mathit{else}\ (=\ (\mathit{if}\ x = B\ \mathit{then}\ 21\ \mathit{else}\ \mathit{if}\ x = C\ \mathit{then}\ 42\ \mathit{else}\ 0)\ 42)$

$\lambda x.\mathit{if}\ x = A\ \mathit{then}\ \top\ \mathit{else}\ \underline{(=\ (\mathit{if}\ x = B\ \mathit{then}\ 21\ \mathit{else}\ \mathit{if}\ x = C\ \mathit{then}\ 42\ \mathit{else}\ 0)}\ 42)$ [10]

$\lambda x.\mathit{if}\ x = A\ \mathit{then}\ \top\ \mathit{else}\ \underline{(\mathit{if}\ x = B\ \mathit{then}\ (=\ 21)\ \mathit{else}\ (=\ (\mathit{if}\ x = C\ \mathit{then}\ 42\ \mathit{else}\ 0)\ 42))}$ [11]

$\lambda x.\mathit{if}\ x = A\ \mathit{then}\ \top\ \mathit{else}\ \mathit{if}\ x = B\ \mathit{then}\ \underline{(21 = 42)}\ \mathit{else}\ (=\ (\mathit{if}\ x = C\ \mathit{then}\ 42\ \mathit{else}\ 0)\ 42)$

$\lambda x.\mathit{if}\ x = A\ \mathit{then}\ \top\ \mathit{else}\ \mathit{if}\ x = B\ \mathit{then}\ \bot\ \mathit{else}\ \underline{(=\ (\mathit{if}\ x = C\ \mathit{then}\ 42\ \mathit{else}\ 0)}\ 42)$ [10]

$\vdots$

$\lambda x.\mathit{if}\ x = A\ \mathit{then}\ \top\ \mathit{else}\ \mathit{if}\ x = B\ \mathit{then}\ \bot\ \mathit{else}\ \mathit{if}\ x = C\ \mathit{then}\ \top\ \mathit{else}\ \bot$

Fig. 3 Computation of $\lambda x.((f\ x) = 42)$. The redexes are underlined. The equation used to rewrite each redex is shown on the right.

$\underline{(\mathit{setOfBalls}\ 2\ s)}$

$\mathit{if}\ \exists x.\exists y.(\underline{(\mathit{colour}\ x)} \wedge (\mathit{colour}\ y) \wedge s = \{(1,x),(2,y)\})\ \mathit{then}\ 0.5^2\ \mathit{else}\ 0$

$\mathit{if}\ \exists x.\exists y.\underline{(((x = \mathit{Blue}) \vee (x = \mathit{Green})) \wedge (\mathit{colour}\ y) \wedge s = \{(1,x),(2,y)\})}\ \mathit{then}\ 0.5^2\ \mathit{else}\ 0$ [12]

$\mathit{if}\ \underline{\exists x.\exists y.(((x = \mathit{Blue}) \wedge (\mathit{colour}\ y) \wedge s = \{(1,x),(2,y)\}) \vee}$
$\underline{((x = \mathit{Green}) \wedge (\mathit{colour}\ y) \wedge s = \{(1,x),(2,y)\}))}\ \mathit{then}\ 0.5^2\ \mathit{else}\ 0$ [15]

$\mathit{if}\ \underline{\exists x.\exists y.((x = \mathit{Blue}) \wedge (\mathit{colour}\ y) \wedge s = \{(1,x),(2,y)\})} \vee$
$\exists x.\exists y.((x = \mathit{Green}) \wedge (\mathit{colour}\ y) \wedge s = \{(1,x),(2,y)\})\ \mathit{then}\ 0.5^2\ \mathit{else}\ 0$ [14]

$\mathit{if}\ \exists y.(\underline{(\mathit{colour}\ y)} \wedge s = \{(1,\mathit{Blue}),(2,y)\}) \vee$
$\exists x.\exists y.((x = \mathit{Green}) \wedge (\mathit{colour}\ y) \wedge s = \{(1,x),(2,y)\})\ \mathit{then}\ 0.5^2\ \mathit{else}\ 0$

$\mathit{if}\ \exists y.\underline{(((y = \mathit{Blue}) \vee (y = \mathit{Green})) \wedge s = \{(1,\mathit{Blue}),(2,y)\})} \vee$
$\exists x.\exists y.((x = \mathit{Green}) \wedge (\mathit{colour}\ y) \wedge s = \{(1,x),(2,y)\})\ \mathit{then}\ 0.5^2\ \mathit{else}\ 0$ [12]

$\mathit{if}\ \underline{\exists y.(((y = \mathit{Blue}) \wedge s = \{(1,\mathit{Blue}),(2,y)\}) \vee ((y = \mathit{Green}) \wedge s = \{(1,\mathit{Blue}),(2,y)\}))} \vee$
$\exists x.\exists y.((x = \mathit{Green}) \wedge (\mathit{colour}\ y) \wedge s = \{(1,x),(2,y)\})\ \mathit{then}\ 0.5^2\ \mathit{else}\ 0$ [15]

$\mathit{if}\ \underline{\exists y.((y = \mathit{Blue}) \wedge s = \{(1,\mathit{Blue}),(2,y)\})} \vee \exists y.((y = \mathit{Green}) \wedge s = \{(1,\mathit{Blue}),(2,y)\}) \vee$
$\exists x.\exists y.((x = \mathit{Green}) \wedge (\mathit{colour}\ y) \wedge s = \{(1,x),(2,y)\})\ \mathit{then}\ 0.5^2\ \mathit{else}\ 0$ [14]

$\mathit{if}\ s = \{(1,\mathit{Blue}),(2,\mathit{Blue})\} \vee \underline{\exists y.((y = \mathit{Green}) \wedge s = \{(1,\mathit{Blue}),(2,y)\})} \vee$
$\exists x.\exists y.((x = \mathit{Green}) \wedge (\mathit{colour}\ y) \wedge s = \{(1,x),(2,y)\})\ \mathit{then}\ 0.5^2\ \mathit{else}\ 0$ [14]

$\mathit{if}\ s = \{(1,\mathit{Blue}),(2,\mathit{Blue})\} \vee s = \{(1,\mathit{Blue}),(2,\mathit{Green})\} \vee$
$\underline{\exists x.\exists y.((x = \mathit{Green}) \wedge (\mathit{colour}\ y) \wedge (s = \{(1,x),(2,y)\}))}\ \mathit{then}\ 0.5^2\ \mathit{else}\ 0$ [14]

$\vdots$

$\mathit{if}\ s = \{(1,\mathit{Blue}),(2,\mathit{Blue})\} \vee s = \{(1,\mathit{Blue}),(2,\mathit{Green})\} \vee$
$s = \{(1,\mathit{Green}),(2,\mathit{Blue})\} \vee s = \{(1,\mathit{Green}),(2,\mathit{Green})\}\ \mathit{then}\ 0.5^2\ \mathit{else}\ 0$

Fig. 4 Computation of $(\mathit{setOfBalls}\ 2\ s)$.

The support $\lambda x.((\mathit{ballsDrawn}\ 2\ \{(1, \mathit{Blue}), (2, \mathit{Green})\}\ x) > 0)$ of the density over lists of balls $(\mathit{ballsDrawn}\ 2\ \{(1, \mathit{Blue}), (2, \mathit{Green})\})$ can be computed in like fashion to yield

$$\lambda x.(x = [(1, \mathit{Blue}), (1, \mathit{Blue})] \vee x = [(1, \mathit{Blue}), (2, \mathit{Green})] \vee \\ x = [(2, \mathit{Green}), (1, \mathit{Blue})] \vee x = [(2, \mathit{Green}), (2, \mathit{Green})]).$$

4.2 Probabilistic Inference

We give in this section a few rules that are needed to support probabilistic inference in Bach. They are all rules for manipulating sums and products.

4.2.1 Sums and Products

In answering probabilistic queries, we are really computing (ratios of) marginalisations of density functions. These expressions usually contain summations of the form $\sum_{x \in s} f(x)$ that we need to evaluate. In the case when s can be efficiently enumerated, we introduce the following function to represent such sums.

$$\mathit{sum} : \{a\} \to (a \to \mathit{Real}) \to \mathit{Real}$$
$$(\mathit{sum}\ \lambda x.(\mathit{if}\ (x = y)\ \mathit{then}\ \top\ \mathit{else}\ \mathbf{w})\ f) = (f\ y) + (\mathit{sum}\ \lambda x.\mathbf{w}\ f)$$
$$(\mathit{sum}\ \lambda x.(\mathit{if}\ (x = y)\ \mathit{then}\ \bot\ \mathit{else}\ \mathbf{w})\ f) = (\mathit{sum}\ \lambda x.\mathbf{w}\ f)$$
$$(\mathit{sum}\ \lambda x.(x = y)\ f) = (f\ y)$$
$$(\mathit{sum}\ \lambda x.((x = y) \vee \mathbf{w})\ f) = (f\ y) + (\mathit{sum}\ \lambda x.\mathbf{w}\ f)$$
$$(\mathit{sum}\ \lambda x.(\mathbf{w} \vee (x = y))\ f) = (f\ y) + (\mathit{sum}\ \lambda x.\mathbf{w}\ f)$$
$$(\mathit{sum}\ \lambda x.(\mathbf{v} \vee \mathbf{w})\ f) = (\mathit{sum}\ \lambda x.\mathbf{v}\ f) + (\mathit{sum}\ \lambda x.\mathbf{w}\ f)$$
$$(\mathit{sum}\ \lambda x.\bot\ f) = 0$$

Thus the term $(\mathit{sum}\ s\ f)$ denotes, in a natural way, the (informal mathematical) expression $\sum_{x \in s} f(x)$. In the definition of *sum* above, we have assumed that the set s can be reduced to the syntactic form of an abstraction over either a nested *if_then_else* or a nested disjunction for which there are no repeated occurrences of the $(x = y)$ tests. Thus, for example, both

$$\lambda x.(x = 1 \vee x = 2 \vee x = 1) \quad \text{and}$$
$$\lambda x.\mathit{if}\ (x = 1)\ \mathit{then}\ \top\ \mathit{else\ if}\ (x = 2)\ \mathit{then}\ \top\ \mathit{else\ if}\ (x = 1)\ \mathit{then}\ \top\ \mathit{else}\ \bot$$

are disallowed. If there are such repeats, it is easy enough to remove them [35, p.189]. The two syntactic forms are sufficiently useful for a wide range of applications.

Example 11 Using *sum*, (7) above can be written as

$$1/K \times (\mathit{sum}\ \lambda s.((\mathit{setOfBalls}\ m\ s) > 0) \\ \lambda s.(\mathit{sum}\ \lambda l.((\mathit{ballsDrawn}\ d\ s\ l) > 0)\ \lambda l.(\mathit{joint}\ d\ (m, s, l, [o_1, \ldots, o_d])))).$$

We need to deal with products as well as summations. For that, we introduce the function $prod : \{a\} \rightarrow (a \rightarrow Real) \rightarrow Real$, which has an analogous definition to that of *sum*. Thus $(prod\ s\ f)$ denotes $\prod_{x \in s} f(x)$ in the natural way.

For convenience, we will use the following shorthand notation

$$\sum_{x \in s} t \equiv (sum\ s\ \lambda x.t) \qquad \prod_{x \in s} t \equiv (prod\ s\ \lambda x.t)$$

in the remainder of the text. Thus, e.g., $\sum_{w \in s_1} \sum_{x \in s_1} \prod_{y \in s_2} \sum_{z \in s_2} (f\ (w, x, y, z))$ denotes the term

$$(sum\ s_1\ \lambda w.(sum\ s_1\ \lambda x.(prod\ s_2\ \lambda y.(sum\ s_2\ \lambda z.(f\ (w, x, y, z)))))).$$

Before moving on, we look at two special cases of summation expressions that will be needed to define the composition functions \$ and § described in Section 2.2. The two functions are needed to solve some computational problems associated with Example 6 and Bayesian tracking (Section 5.2).

In the case when the set s in $\sum_{x \in s} f(x)$ cannot be efficiently enumerated but there is an (intensional) predicate representation for it so that set membership can be computed easily, we can use the following function *sum2* to compute the summation.

$$sum2 : \{a\} \rightarrow (a \rightarrow Real) \rightarrow Real$$

$$(sum2\ s\ \lambda x.if\ x = u\ then\ v\ else\ \boldsymbol{w}) = (if\ (s\,u)\ then\ v\ else\ 0) + (sum2\ s\ \lambda x.\boldsymbol{w}) \tag{18}$$

$$(sum2\ s\ \lambda x.0) = 0 \tag{19}$$

Here it is assumed that the second argument to *sum2* has the syntactic form of an abstraction over a nested *if_then_else* for which there are no repeated occurrences of the tests $x = u$. Similar remarks apply to *sum3* below.

We will also need a function *sum3* that computes $\sum_x f(x)$ in the case when x ranges over the whole domain of f.

$$sum3 : (a \rightarrow Real) \rightarrow Real$$

$$(sum3\ \lambda x.if\ x = u\ then\ v\ else\ \boldsymbol{w}) = v + (sum3\ \lambda x.\boldsymbol{w}) \tag{20}$$

$$(sum3\ \lambda x.0) = 0 \tag{21}$$

Example 12 The discrete case of the two composition functions \$ and § described in Section 2.2 can be defined in the logic as follows.

$$\$: Density\ b \rightarrow (b \rightarrow c) \rightarrow Density\ c$$

$$f\ \$\ g = \lambda z.(sum2\ \lambda y.((g\ y) = z)\ f) \tag{22}$$

$$\S : Density\ b \rightarrow (b \rightarrow Density\ c) \rightarrow Density\ c$$

$$f\ \S\ g = \lambda z.(sum3\ \lambda y.((f\ y) \times ((g\ y)\ z))). \tag{23}$$

4.2.2 Variable Elimination

In computing marginalisations of a given joint density, we can exploit the factorization of the joint to move products outside sums, a transformation procedure commonly known as *variable elimination*. The following rule is essentially what we need.

$$\sum_{y \in s} t_1 \times \cdots \times t_n \;=\; t_i \times \sum_{y \in s} t_1 \times \cdots \times t_{i-1} \times t_{i+1} \times \cdots \times t_n \tag{24}$$

-- if y is not free in t_i.

Example 13 Fig. 5 shows how a term of the form

$$\sum_{x \in s_1} \sum_{y \in s_2} \sum_{z \in s_3} (g_1\ x) \times (g_2\ (x, y)) \times (g_3\ (y, z))$$

can be simplified using Equation (24).

$$\sum_{x \in s_1} \sum_{y \in s_2} \underline{\sum_{z \in s_3} (g_1\ x) \times (g_2\ (x, y)) \times (g_3\ (y, z))}$$

$$\sum_{x \in s_1} \underline{\sum_{y \in s_2} (g_1\ x) \times \sum_{z \in s_3} (g_2\ (x, y)) \times (g_3\ (y, z))}$$

$$\sum_{x \in s_1} (g_1\ x) \times \sum_{y \in s_2} \underline{\sum_{z \in s_3} (g_2\ (x, y)) \times (g_3\ (y, z))}$$

$$\sum_{x \in s_1} (g_1\ x) \times \sum_{y \in s_2} (g_2\ (x, y)) \times \sum_{z \in s_3} (g_3\ (y, z))$$

Fig. 5 Computation of $\sum_{x \in s_1} \sum_{y \in s_2} \sum_{z \in s_3} (g_1\ x) \times (g_2\ (x, y)) \times (g_3\ (y, z))$

In contrast to the standard variable-elimination algorithm [60], we move a summation inside a product of factors one factor at a time instead of multiple factors at a time. Further, unlike the standard algorithm, actual summations over a variable are usually delayed as much as possible courtesy of the leftmost (lazy) redex selection rule.

4.2.3 Lifted Inference

Many existing inference algorithms for first-order probabilistic models employ the strategy of first grounding out the model before applying standard inference techniques. There are a few problems with this approach. First, the grounding step can only be done if the domain of the model is finite, an unworkable assumption in many applications. Further, for many kinds of queries, it is unnecessary to consider all the random variables in the model. Lifted first-order probabilistic inference algorithms avoid the grounding step by working directly with groups of random variables, exploiting opportunities to simplify and/or eliminate sets of random variables all in one go. Examples of such algorithms appear in [50] and [11].

Building on [11,12], we develop a set of tools to support lifted probabilistic inference in this section. We will use the factor graph shown in Fig. 6 to motivate the key ideas.

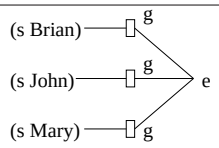


Fig. 6 A fragment of the factor graph shown in Fig. 1

Let $P = \{Brian, John, Mary\}$. The joint for the factor graph is defined as follows:

$$joint : ((Person \to \Omega) \times \Omega) \to Real$$
$$(joint\ (s, e)) = \prod_{x \in P} (g\ (s\ x)\ e).$$

Suppose we want to compute $Pr(e = \top \mid (s\ Mary) = \top)$. The (informal) mathematical expression that we need to evaluate is

$$\frac{1}{K} \sum_{(s\ Brian) \in \{\top, \bot\}} \sum_{(s\ John) \in \{\top, \bot\}} (g\ (s\ Brian)\ \top) \times (g\ (s\ John)\ \top) \times (g\ \top\ \top), \quad (25)$$

where K is a normalisation constant and the two terms $(s\ Brian)$ and $(s\ John)$ are treated as variables ranging over the set $\{\top, \bot\}$. The need to treat $(s\ Brian)$ and $(s\ John)$ as variables, which they are clearly not, is the main technical problem to be addressed in encoding (25) in the logic. A compact and logically clean way of writing (25) in Bach is

$$\frac{1}{K} \sum_{s \in \{\top, \bot\}^{\{John, Brian\}}} (g\ (s\ Brian)\ \top) \times (g\ (s\ John)\ \top) \times (g\ \top\ \top), \quad (26)$$

where $\{\top, \bot\}^{\{John, Brian\}}$ is the set of all functions from $\{John, Brian\}$ to $\{\top, \bot\}$; i.e.,

$$\begin{aligned} \{\top, \bot\}^{\{John, Brian\}} = \{\ & \lambda x.if\ x = John\ then\ \top\ else\ if\ x = Brian\ then\ \top\ else\ v_d, \\ & \lambda x.if\ x = John\ then\ \top\ else\ if\ x = Brian\ then\ \bot\ else\ v_d, \\ & \lambda x.if\ x = John\ then\ \bot\ else\ if\ x = Brian\ then\ \top\ else\ v_d, \\ & \lambda x.if\ x = John\ then\ \bot\ else\ if\ x = Brian\ then\ \bot\ else\ v_d\ \}, \end{aligned}$$

where v_d is some default value, say, $\bot$. To see that (26) correctly captures the intended meaning of (25), observe that (26) can be expanded out to the following term

$$\begin{aligned} 1/K(\quad & (\lambda s.(g\ (s\ Brian)\ \top) \times (g\ (s\ John)\ \top) \times (g\ \top\ \top) \\ & \qquad \lambda x.if\ x = John\ then\ \top\ else\ if\ x = Brian\ then\ \top\ else\ v_d) \\ & + (\lambda s.(g\ (s\ Brian)\ \top) \times (g\ (s\ John)\ \top) \times (g\ \top\ \top) \\ & \qquad \lambda x.if\ x = John\ then\ \top\ else\ if\ x = Brian\ then\ \bot\ else\ v_d) \\ & + (\lambda s.(g\ (s\ Brian)\ \top) \times (g\ (s\ John)\ \top) \times (g\ \top\ \top) \\ & \qquad \lambda x.if\ x = John\ then\ \bot\ else\ if\ x = Brian\ then\ \top\ else\ v_d) \\ & + (\lambda s.(g\ (s\ Brian)\ \top) \times (g\ (s\ John)\ \top) \times (g\ \top\ \top) \\ & \qquad \lambda x.if\ x = John\ then\ \bot\ else\ if\ x = Brian\ then\ \bot\ else\ v_d)), \end{aligned} \quad (27)$$

which can be simplified to the exact term we need:

$$1/K((g\ \top\ \top) \times (g\ \top\ \top) \times (g\ \top\ \top)\ +\ (g\ \bot\ \top) \times (g\ \top\ \top) \times (g\ \top\ \top) \\ +\ (g\ \top\ \top) \times (g\ \bot\ \top) \times (g\ \top\ \top)\ +\ (g\ \bot\ \top) \times (g\ \bot\ \top) \times (g\ \top\ \top)). \quad (28)$$

Fig. 7 shows how the first term in the summation in (27) is reduced to the first term in the summation in (28). The other terms are reduced in a similar way.

$$\underline{(\lambda s.(g\ (s\ Brian)\ \top) \times (g\ (s\ John)\ \top) \times (g\ \top\ \top)} \\ \underline{\lambda x.if\ x = John\ then\ \top\ else\ if\ x = Brian\ then\ \top\ else\ v_d)} \quad [13]$$

$$(g\ \underline{(\lambda x.if\ x = John\ then\ \top\ else\ if\ x = Brian\ then\ \top\ else\ v_d\ Brian)}\ \top) \\ \times\ (g\ (\lambda x.if\ x = John\ then\ \top\ else\ if\ x = Brian\ then\ \top\ else\ v_d\ John)\ \top) \times (g\ \top\ \top) \quad [13]$$

$$(g\ (if\ \underline{Brian = John}\ then\ \top\ else\ if\ Brian = Brian\ then\ \top\ else\ v_d)\ \top) \\ \times\ (g\ (\lambda x.if\ x = John\ then\ \top\ else\ if\ x = Brian\ then\ \top\ else\ v_d\ John)\ \top) \times (g\ \top\ \top)$$

$$(g\ \underline{(if\ \bot\ then\ \top\ else\ if\ Brian = Brian\ then\ \top\ else\ v_d)}\ \top) \\ \times\ (g\ (\lambda x.if\ x = John\ then\ \top\ else\ if\ x = Brian\ then\ \top\ else\ v_d\ John)\ \top) \times (g\ \top\ \top) \quad [9]$$

$$\vdots$$

$$(g\ \top\ \top) \times (g\ \underline{(\lambda x.if\ x = John\ then\ \top\ else\ if\ x = Brian\ then\ \top\ else\ v_d\ John)}\ \top) \times (g\ \top\ \top) \quad [13]$$

$$(g\ \top\ \top) \times (g\ (if\ \underline{John = John}\ then\ \top\ else\ if\ John = Brian\ then\ \top\ else\ v_d)\ \top) \times (g\ \top\ \top)$$

$$\vdots$$

$$(g\ \top\ \top) \times (g\ \top\ \top) \times (g\ \top\ \top)$$

Fig. 7 Computation of the first term in the summation in (27)

The above discussion motivates the need to sum over sets of functions. Let X and Y be two arbitrary finite sets. The set of all functions from X to Y is denoted by Y^X. Suppose $X = \{x_1, \ldots, x_k\}$ and y_d is an arbitrarily chosen default element in Y. We define Y^X in Bach as follows:

$$\{\, \lambda x.if\ x = x_1\ then\ v_1\ else \cdots else\ if\ x = x_k\ then\ v_k\ else\ y_d \mid v_i \in Y \,\}. \quad (29)$$

The set $\{\top, \bot\}^{\{John, Brian\}}$ given above is an example of such a set. We can recursively apply (29) to arrive at, for example, $\{\top, \bot\}^{Y^X}$, which is the set of all functions with type $X \to Y \to \{\top, \bot\}$. In practice, the sets X and Y in Y^X need only be specified intensionally and their enumeration is taken care of automatically by Bach. The set Y^X, which can be large, is also only enumerated by Bach when strictly necessary. The key techniques needed to achieve all that are explained earlier in Examples 9 and 10.

We are now in a position to present two key ideas needed to support lifted inference. First-order versions of these results were presented earlier in [11]. We clarify some of those results here. Proofs of correctness are deferred to the appendix.

Inversion Elimination The central idea behind inversion elimination, which was introduced in [50] and subsequently formalised in [11], can be motivated by the following

equation

$$\sum_{f\in Y^X}\prod_{x\in X}(g\ (f\ x)) = \prod_{x\in X}\sum_{f\in Y^{\{x\}}}(g\ (f\ x)), \tag{30}$$

where X and Y are two finite sets and g some real-valued function on Y. It is clear that the RHS of (30) is computationally more tractable than the LHS. To get an intuition for (30), consider the case when $X = \{x_1, x_2\}$ and $Y = \{\top, \bot\}$. Expanding out the RHS of (30), we have

$$\begin{aligned}
\text{RHS of (30)} &= \Big(\sum_{f\in Y^{\{x_1\}}}(g\ (f\ x_1))\Big) \times \Big(\sum_{f\in Y^{\{x_2\}}}(g\ (f\ x_2))\Big)\\
&= ((g\ (f_{1\top}\ x_1)) + (g\ (f_{1\bot}\ x_1))) \times ((g\ (f_{2\top}\ x_2)) + (g\ (f_{2\bot}\ x_2)))\\
&= (g\ (f_{1\top}\ x_1)) \times (g\ (f_{2\top}\ x_2)) + (g\ (f_{1\top}\ x_1)) \times (g\ (f_{2\bot}\ x_2)) +\\
&\quad (g\ (f_{1\bot}\ x_1)) \times (g\ (f_{2\top}\ x_2)) + (g\ (f_{1\bot}\ x_1)) \times (g\ (f_{2\bot}\ x_2))\\
&= \sum_{f\in Y^X}\prod_{x\in X}(g\ (f\ x)),
\end{aligned}$$

where f_{iy} is a shorthand for $\lambda x.\mathit{if}\ x = x_i\ \mathit{then}\ y\ \mathit{else}\ \bot$. The basic argument above works for any finite sets X and Y. Indeed, (30) can be significantly generalised, leading to the following formalisation.

Let t be a real-valued term such that each free occurrence of the variable f in t occurs in a subterm of the form $(f\ x_1\ \ldots\ x_k)$, where $x_1, \ldots, x_k$ are free occurrences in t of these variables and $1 \le k \le n$, and $X_1, \ldots, X_n, Y$ be finite sets. Then

$$\sum_{f\in Y^{X_n^{\cdot^{\cdot^{\cdot^{\cdot^{X_1}}}}}}}\prod_{x_1\in X_1}\cdots\prod_{x_k\in X_k} t = \prod_{x_1\in X_1}\cdots\prod_{x_k\in X_k}\sum_{f\in Y^{X_n^{\cdot^{\cdot^{\cdot^{\cdot^{X_{k+1}^{\{x_k\}}}}}}}}}\cdots^{\cdot^{\{x_1\}}} t. \tag{31}$$

Example 14 Let X, Y, Z be non-empty finite sets, τ the type of elements in Z, and $f : \tau \to \tau \to \mathit{Real}$ an arbitrary function. Here are some special cases of (31).

$$\sum_{p\in Z^X}\prod_{x\in X}(f\ (p\ x)\ (p\ x)) = \prod_{x\in X}\sum_{p\in Z^{\{x\}}}(f\ (p\ x)\ (p\ x))$$

$$\sum_{q\in Z^{Y^X}}\prod_{x\in X}\prod_{y\in Y}(f\ (p\ x)\ (q\ x\ y)) = \prod_{x\in X}\prod_{y\in Y}\sum_{q\in Z^{\{y\}^{\{x\}}}}(f\ (p\ x)\ (q\ x\ y))$$

$$\sum_{q\in Z^{Y^X}}\prod_{x\in X}\prod_{y\in Y}\prod_{z\in Y}(f\ (q\ x\ y)\ (q\ x\ z)) = \prod_{x\in X}\sum_{q\in Z^{Y^{\{x\}}}}\prod_{y\in Y}\prod_{z\in Y}(f\ (q\ x\ y)\ (q\ x\ z)).$$

Here is an example that illustrates multiple applications of (31).

$$\begin{aligned}
&\sum_{p\in Z^{Y^X}}\sum_{q\in Z^{Y^X}}\prod_{x\in X}\prod_{y\in Y}(f\ (p\ x\ y)\ (q\ x\ y))\\
&\qquad = \sum_{p\in Z^{Y^X}}\prod_{x\in X}\prod_{y\in Y}\sum_{q\in Z^{\{y\}^{\{x\}}}}(f\ (p\ x\ y)\ (q\ x\ y))\\
&\qquad = \prod_{x\in X}\prod_{y\in Y}\sum_{p\in Z^{\{y\}^{\{x\}}}}\sum_{q\in Z^{\{y\}^{\{x\}}}}(f\ (p\ x\ y)\ (q\ x\ y)).
\end{aligned}$$

Here are some examples of terms that cannot be simplified using (31).

$$\sum_{p\in Z^X}\prod_{x\in X}\prod_{y\in X}(f\ (p\ x)\ (p\ y))$$

$$\sum_{p\in Z^X}\prod_{x\in X}(f\ (p\ x)\ (p\ c)) \qquad \text{– where } c \text{ is an element in } X$$

$$\sum_{p\in Z^X}\prod_{x\in X}(f\ (p\ x)\ (p\ x))\sum_{x\in X}(f\ (p\ x)\ (p\ x)).$$

Counting Elimination The basic idea behind counting elimination can be illustrated by considering the problem of simplifying an expression like

$$\sum_{p\in\{\top,\bot\}^X}\prod_{x\in X}(f\ (p\ x)),$$

where X is a finite set with its cardinality denoted by $|X|$ and $f : \Omega \to \mathit{Real}$ is an arbitrary function. Given any fixed $p \in \{\top,\bot\}^X$, it is easy to see that $\prod_{x\in X}(f\ (p\ x))$ reduces to the expression $(f\ \top)^i(f\ \bot)^{|X|-i}$, where p maps i elements in X to $\top$ and the rest to $\bot$. Observing that there are $\binom{|X|}{i}$ functions in $\{\top,\bot\}^X$ that map exactly i elements in X to $\top$, we have

$$\sum_{p\in\{\top,\bot\}^X}\prod_{x\in X}(f\ (p\ x)) \;=\; \sum_{i=0}^{|X|}\binom{|X|}{i}(f\ \top)^i(f\ \bot)^{|X|-i},$$

where the RHS is computationally more tractable than the LHS. The kind of reasoning we have just gone through can be generalised, leading to the counting-elimination result encapsulated in (32) below.

We first establish some notation in preparation for the result. If $m, n \geq 1$, let

$$M_m^n = \{(k_1,\ldots,k_m) \mid k_1+\cdots+k_m = n, \text{ where } k_i \geq 0, \text{ for } i = 1,\ldots,m\}.$$

If $\pi \equiv (k_1,\ldots,k_m) \in M_m^n$, let

$$C_\pi = \binom{n}{k_1,\ldots,k_m}.$$

If Y is a set of cardinality m, $\phi : Y \to \{1,\ldots,m\}$ is a fixed, but arbitrary, bijection, and $\pi \equiv (k_1,\ldots,k_m) \in M_m^n$, then $\mu_\pi : Y \to \mathbb{N}$ is defined by

$$\mu_\pi(y) = k_{\phi(y)},$$

for $y \in Y$.

Let t be a real-valued term such that each free occurrence of the variable f_i in t occurs in a subterm of the form $(f_i\ x_{i1})$ or $\ldots$ or $(f_i\ x_{ir_i})$, where $x_{i1},\ldots,x_{ir_i}$ are free occurrences in t of these variables that do not have any free occurrences in t other than these, for $i = 1,\ldots,p$. Let $y_{i1},\ldots,y_{ir_i}$ be variables not appearing in t, for $i = 1,\ldots,p$.

Also let X_i be a set of cardinality n_i and Y_i a set of cardinality m_i, for $i = 1, \ldots, p$. Then

$$\begin{aligned}
&\sum_{f_1 \in {Y_1}^{X_1}} \cdots \sum_{f_p \in {Y_p}^{X_p}} \prod_{x_{11} \in X_1} \cdots \prod_{x_{1r_1} \in X_1} \cdots \prod_{x_{p1} \in X_p} \cdots \prod_{x_{pr_p} \in X_p} t \\
&= \sum_{\pi_1 \in M_{m_1}^{n_1}} \cdots \sum_{\pi_p \in M_{m_p}^{n_p}} \prod_{i=1}^{p} C_{\pi_i} \prod_{y_{11} \in Y_1} \cdots \prod_{y_{1r_1} \in Y_1} \cdots \prod_{y_{p1} \in Y_p} \cdots \prod_{y_{pr_p} \in Y_p} \\
&\qquad t\{(f_1\ x_{11})/y_{11}, \ldots, (f_p\ x_{pr_p})/y_{pr_p}\}^{\mu_{\pi_1}(y_{11}) \ldots \mu_{\pi_1}(y_{1r_1}) \ldots \mu_{\pi_p}(y_{p1}) \ldots \mu_{\pi_p}(y_{pr_p})}.
\end{aligned} \tag{32}$$

(The notation $t\{(f_1\ x_{11})/y_{11}, \ldots, (f_p\ x_{pr_p})/y_{pr_p}\}$ means that all occurrences in t of $(f_1\ x_{11})$, where the occurrences of f_1 and x_{11} are free, are replaced by y_{11}, and so on.) The above captures the essence of Theorem 15.1 in [12] but is rather more general.

Example 15 Let X, Y and Z be three arbitrary non-empty finite sets, τ the type of the elements in Z, $f : \tau \to \tau \to \mathit{Real}$ and $g : \tau \to \tau \to \tau \to \mathit{Real}$ two arbitrary functions. Here are some special cases of (32):

$$\sum_{p \in Z^X} \prod_{x \in X} \prod_{y \in X} (f\ (p\ x)\ (p\ y)) = \sum_{\pi \in M_{|Z|}^{|X|}} C_\pi \prod_{x' \in Z} \prod_{y' \in Z} (f\ x'\ y')^{(\mu_\pi\ x')(\mu_\pi\ y')}$$

$$\sum_{p \in Z^X} \prod_{x \in X} (f\ (p\ x)\ (p\ x)) = \sum_{\pi \in M_{|Z|}^{|X|}} C_\pi \prod_{x' \in Z} (f\ x'\ x')^{(\mu_\pi\ x')}$$

$$\sum_{p \in Z^X} \prod_{x \in X} \prod_{y \in Y} (f\ (p\ x)\ (q\ y)) = \sum_{\pi \in M_{|Z|}^{|X|}} C_\pi \prod_{x' \in Z} \Big(\prod_{y \in Y} (f\ x'\ (q\ y)) \Big)^{(\mu_\pi\ x')}$$

$$\begin{aligned}
&\sum_{p \in Z^X} \sum_{q \in Z^Y} \prod_{x \in X} \prod_{y \in X} \prod_{z \in Y} (g\ (p\ x)\ (q\ z)\ (p\ y)) \ = \\
&\qquad \sum_{\pi \in M_{|Z|}^{|X|}} \sum_{\nu \in M_{|Z|}^{|Y|}} C_\pi C_\nu \prod_{x' \in Z} \prod_{y' \in Z} \prod_{z' \in Z} (g\ x'\ z'\ y')^{(\mu_\pi\ x')(\mu_\pi\ y')(\mu_\nu\ z')}
\end{aligned}$$

$$\begin{aligned}
&\sum_{p \in Z^X} \prod_{x \in X} \prod_{y \in X} (f\ (p\ x)\ (p\ y)) \sum_{p \in Z^Y} \prod_{x \in Y} \prod_{y \in Z} (f\ (p\ x)\ y) \ = \\
&\qquad \sum_{\pi \in M_{|Z|}^{|X|}} C_\pi \prod_{x' \in Z} \prod_{y' \in Z} \Big((f\ x'\ y') \sum_{p \in Z^Y} \prod_{x \in Y} \prod_{y \in Z} (f\ (p\ x)\ y) \Big)^{(\mu_\pi\ x')(\mu_\pi\ y')}.
\end{aligned}$$

The third and fifth equations above are examples of equations that cannot be handled in the formulation of [12].

Here are some examples of expressions that cannot be simplified using (32).

$$\sum_{p \in Z^X} \prod_{x \in X} (f\ (p\ x)\ (p\ y))$$

$$\sum_{p \in Z^X} \prod_{x \in X} (f\ (p\ x)\ (p\ c)) \qquad \text{– where } c \text{ is an element of } X$$

$$\sum_{p \in Z^X} \prod_{x \in X} \prod_{y \in Y} (f\ (p\ x)\ (q\ y)) \prod_{x \in X} (f\ (q\ y)\ (p\ x))$$

Auxiliary Operation Equations (31) and (32) are powerful when we can apply them, but expressions that can be simplified using those two equations may not always be available in the form required. In both [50] and [12], auxiliary operations like shattering are introduced to reorganise expressions into desired forms. Here we introduce one such useful equation:

$$\sum_{f \in Z^X} t_1 \times t_2 = \left(\sum_{f \in Z^Y} t_1 \right) \times \left(\sum_{f \in Z^{X \setminus Y}} t_2 \right) \tag{33}$$

if there exists $Y \subset X$ such that

1. for each subterm of the form $(f\ t)$ in t_1, either $t \in Y$ or t is a variable bound in t_1 that can only be substituted with an element in Y;
2. for each subterm of the form $(f\ t)$ in t_2, either $t \notin Y$ or t is a variable bound in t_2 that can only be substituted with an element in $X \setminus Y$.

For common cases encountered in practice, the set Y can be automatically determined.

Equations (31)-(33) are implemented as system-level equations in Bach. We will see how they can be used to solve the problem originally posed in Example 7 in Section 4.3.2. Before proceeding, we note that the formulation of (31)-(33) and associated concepts all involve natural and non-trivial use of higher-order functions (functions that take functions as arguments and/or return functions as values). In formalising lifted inference, these higher-order concepts can appear either in the semantics of an inference system (as in [50] and [12]) or directly in the syntax and proof theory of a logical system (as here). The advantage of the latter is that it makes a complete, declarative implementation of lifted inference significantly easier.

4.3 Examples of Probabilistic Inference

Having laid the necessary groundwork, we are now in a position to show in detail how the example problems given in Section 3 can be solved.

4.3.1 TV Agent

We now see how the value of the term

$$((\mathit{choice}\ \$\ \mathit{tv\_guide})\ \S\ \mathit{likes})$$

described in Example 6 can be computed. The computation is somewhat complicated. To make it easier to understand, the computation is broken up into two steps. Fig. 8 gives the computation of $(\mathit{choice}\ \$\ \mathit{tv\_guide})$ that gives the answer

$$\begin{aligned}&\lambda y.\mathit{if}\ y = (\text{“}\mathit{The\ Bill}\text{”}, 50, \mathit{Drama}, M, \text{“}\mathit{Sun\ Hill}\ldots\text{”})\ \mathit{then}\ 0.2\ \mathit{else}\\ &\quad \mathit{if}\ y = (\text{“}\mathit{Seinfeld}\text{”}, 30, \mathit{Sitcom}, PG, \text{“}\mathit{Kramer}\ldots\text{”})\ \mathit{then}\ 0.8\ \mathit{else}\ 0,\end{aligned}$$

which is a term of type *Density Program*. Then the second part of the computation in Fig. 9 is that of

$$\begin{aligned}&((\lambda y.\mathit{if}\ y = (\text{“}\mathit{The\ Bill}\text{”}, \ldots)\ \mathit{then}\ 0.2\\ &\qquad\qquad \mathit{else\ if}\ y = (\text{“}\mathit{Seinfeld}\text{”}, \ldots)\ \mathit{then}\ 0.8\ \mathit{else}\ 0)\ \S\ \mathit{likes}),\end{aligned}$$

$\underline{(choice\ \$\ tv\_guide)}$ [22]

$\lambda z.(sum2\ \lambda y.(\underline{(tv\_guide\ y)} = z)\ choice)$ [3]

$\lambda z.(sum2\ \lambda y.((if\ y = ((20,7,2007),(11,30),Win)\ then\ \ldots) = z)\ \underline{choice})$ [2]

$\lambda z.(\underline{sum2\ \lambda y.((if\ y = ((20,7,2007),(11,30),Win)\ then\ \ldots) = z)}$
$\underline{\lambda x.if\ x = ((21,7,2007),(19,30),ABC)\ then\ 0.8\ \ldots})$ [18]

$\vdots$

$\lambda z.(if\ z = (\text{“}Seinfeld\text{”},\ldots)\ then\ 0.8\ else\ 0$
$+\ (\underline{sum2\ \lambda y.((if\ y = ((20,7,2007),(11,30),Win)\ then\ \ldots) = z)}$
$\underline{\lambda x.if\ x = ((21,7,2007),(20,30),ABC)\ then\ 0.2\ \ldots})$ [18]

$\vdots$

$\lambda z.(if\ z = (\text{“}Seinfeld\text{”},\ldots)\ then\ 0.8\ else\ 0$
$+\ if\ z = (\text{“}The\ Bill\text{”},\ldots)\ then\ 0.2\ else\ 0$
$+\ \underline{(sum2\ \lambda y.((if\ y = ((20,7,2007),(11,30),Win)\ then\ \ldots) = z)\ \lambda x.0)}$ [19]

$\lambda z.(if\ z = (\text{“}Seinfeld\text{”},\ldots)\ then\ 0.8\ else\ 0$
$+\ \underline{if\ z = (\text{“}The\ Bill\text{”},\ldots)\ then\ 0.2\ else\ 0\ +\ 0)}$

$\lambda z.(\underline{if\ z = (\text{“}Seinfeld\text{”},\ldots)\ then\ 0.8\ else\ 0\ +\ if\ z = (\text{“}The\ Bill\text{”},\ldots)\ then\ 0.2\ else\ 0)}$ [10]

$\vdots$

$\lambda z.if\ z = (\text{“}The\ Bill\text{”},\ldots)\ then\ 0.2\ else\ if\ z = (\text{“}Seinfeld\text{”},\ldots)\ then\ 0.8\ else\ 0$

Fig. 8 Computation of $(choice\ \$\ tv\_guide)$

the answer of which can be further simplified to

$$\lambda z.if\ z = \top\ then\ 0.76\ else\ if\ z = \bot\ then\ 0.24\ else\ 0$$

using the technique for removing duplicates in a set given in [35, p.189].

4.3.2 Epidemics

Consider now the problem of computing, in the context of Example 7,

$$Pr(e = \top \mid (h\ Brian) = \top, (h\ John) = \top, (h\ Mary) = \top, (h\ Mike) = \bot). \tag{34}$$

Let *vst* (evidence set) denote $\{Brian, John, Mary, Mike\}$ and *othr* denote $(all \setminus vst)$. The term we need to evaluate is

$$\frac{1}{K} \sum_{s \in 2^{all}} \sum_{h \in 2^{othr}} (f\ (s\ Mike)\ \bot)(g\ (s\ Mike)\ \top) \Big(\prod_{x \in vst \setminus \{Mike\}} (f\ (s\ x)\ \top)(g\ (s\ x)\ \top) \Big) \Big(\prod_{x \in all \setminus vst} (f\ (s\ x)\ (h\ x))(g\ (s\ x)\ \top) \Big), \tag{35}$$

$$\underline{((\lambda y.\mathit{if}\ y = (\text{“}\mathit{The\ Bill}\text{”},\ldots)\ \mathit{then}\ 0.2\ \mathit{else\ if}\ y = (\text{“}\mathit{Seinfeld}\text{”},\ldots)\ \mathit{then}\ 0.8\ \mathit{else}\ 0)\ \S\ \mathit{likes})} \quad [23]$$

$$\lambda z.(\mathit{sum3}\ \lambda y.(\underline{((\lambda y.\mathit{if}\ y = (\text{“}\mathit{The\ Bill}\text{”},\ldots)\ \mathit{then}\ 0.2\ \mathit{else}\ \mathit{if}\ y = (\text{“}\mathit{Seinfeld}\text{”},\ldots)\ \mathit{then}\ 0.8\ \mathit{else}\ 0)\ y)} \times ((\mathit{likes}\ y)\ z))) \quad [13]$$

$$\lambda z.(\mathit{sum3}\ \lambda y.(\underline{(\mathit{if}\ y = (\text{“}\mathit{The\ Bill}\text{”},\ldots)\ \mathit{then}\ 0.2\ \mathit{else}\ \mathit{if}\ y = (\text{“}\mathit{Seinfeld}\text{”},\ldots)\ \mathit{then}\ 0.8\ \mathit{else}\ 0)} \times ((\mathit{likes}\ y)\ z))) \quad [10]$$

$$\vdots$$

$$\lambda z.(\mathit{sum3}\ \lambda y.(\mathit{if}\ y = (\text{“}\mathit{The\ Bill}\text{”},\ldots)\ \mathit{then}\ 0.2 \times (\underline{(\mathit{likes}\ (\text{“}\mathit{The\ Bill}\text{”},\ldots))}\ z)\ \mathit{else}\ ((\mathit{if}\ y = (\text{“}\mathit{Seinfeld}\text{”},\ldots)\ \mathit{then}\ 0.8\ \mathit{else}\ 0) \times ((\mathit{likes}\ y)\ z)))) \quad [4]$$

$$\vdots$$

$$\lambda z.(\mathit{sum3}\ \lambda y.(\mathit{if}\ y = (\text{“}\mathit{The\ Bill}\text{”},\ldots)\ \mathit{then}\ 0.2 \times \underline{(\lambda y.\mathit{if}\ y = \top\ \mathit{then}\ 0.2\ \mathit{else\ if}\ y = \bot\ \mathit{then}\ 0.8\ \mathit{else}\ 0\ \ z)}\ \mathit{else}\ ((\mathit{if}\ y = (\text{“}\mathit{Seinfeld}\text{”},\ldots)\ \mathit{then}\ 0.8\ \mathit{else}\ 0) \times ((\mathit{likes}\ y)\ z)))) \quad [13]$$

$$\lambda z.(\mathit{sum3}\ \lambda y.(\mathit{if}\ y = (\text{“}\mathit{The\ Bill}\text{”},\ldots)\ \mathit{then}\ \underline{0.2 \times (\mathit{if}\ z = \top\ \mathit{then}\ 0.2\ \mathit{else\ if}\ z = \bot\ \mathit{then}\ 0.8\ \mathit{else}\ 0)}\ \mathit{else}\ ((\mathit{if}\ y = (\text{“}\mathit{Seinfeld}\text{”},\ldots)\ \mathit{then}\ 0.8\ \mathit{else}\ 0) \times ((\mathit{likes}\ y)\ z)))) \quad [10]$$

$$\vdots$$

$$\lambda z.(\mathit{sum3}\ \lambda y.(\mathit{if}\ y = (\text{“}\mathit{The\ Bill}\text{”},\ldots)\ \mathit{then}\ (\mathit{if}\ z = \top\ \mathit{then}\ 0.04\ \mathit{else\ if}\ z = \bot\ \mathit{then}\ 0.16\ \mathit{else}\ 0)\ \mathit{else}\ (\underline{(\mathit{if}\ y = (\text{“}\mathit{Seinfeld}\text{”},\ldots)\ \mathit{then}\ 0.8\ \mathit{else}\ 0) \times}((\mathit{likes}\ y)\ z)))) \quad [10]$$

$$\vdots$$

$$\lambda z.(\underline{\mathit{sum3}\ \lambda y.(\mathit{if}\ y = (\text{“}\mathit{The\ Bill}\text{”},\ldots)\ \mathit{then}\ (\mathit{if}\ z = \top\ \mathit{then}\ 0.04\ \mathit{else\ if}\ z = \bot\ \mathit{then}\ 0.16\ \mathit{else}\ 0)\ \mathit{else\ if}\ y = (\text{“}\mathit{Seinfeld}\text{”},\ldots)\ \mathit{then}\ (\mathit{if}\ z = \top\ \mathit{then}\ 0.72\ \mathit{else\ if}\ z = \bot\ \mathit{then}\ 0.08\ \mathit{else}\ 0)\ \mathit{else}\ 0))} \quad [20]$$

$$\vdots$$

$$\lambda z.(\underline{(\mathit{if}\ z = \top\ \mathit{then}\ 0.04\ \mathit{else\ if}\ z = \bot\ \mathit{then}\ 0.16\ \mathit{else}\ 0) +\ (\mathit{if}\ z = \top\ \mathit{then}\ 0.72\ \mathit{else\ if}\ z = \bot\ \mathit{then}\ 0.08\ \mathit{else}\ 0))} \quad [10]$$

$$\vdots$$

$$\lambda z.\mathit{if}\ z = \top\ \mathit{then}\ 0.76\ \mathit{else\ if}\ z = \bot\ \mathit{then}\ 0.24\ \mathit{else\ if}\ z = \top\ \mathit{then}\ 0.04\ \mathit{else\ if}\ z = \bot\ \mathit{then}\ 0.16\ \mathit{else}\ 0$$

Fig. 9 Computation of $((\lambda y.\mathit{if}\ y = (\text{“}\mathit{The\ Bill}\text{”},\ldots)\ \mathit{then}\ 0.8\ \mathit{else}\ \ldots) \S \mathit{likes})$

where 2 is $\{\top, \bot\}$, K is a normalisation constant and the summand is just the joint

$$\prod_{x \in all} (f\ (s\ x)\ (h\ x)) \times (g\ (s\ x)\ e)$$

expanded out for the partition of *all* into $\{Mike\}$, $(vst \setminus \{Mike\})$, and *othr* and instantiated with the evidence

$$(h\ Brian) = \top,\ (h\ John) = \top,\ (h\ Mary) = \top,\ \text{and}\ (h\ Mike) = \bot.$$

Fig. 10 shows how the main summation term in (35) can be evaluated efficiently using lifted-inference techniques. Note the drastic improvement achieved: the complexity of the first line is $O(2^{|all|+|othr|}|all|)$, whereas that of the seventh line is $O(|vst|)$.

The normalisation constant K contains two terms corresponding to when $e = \top$ and $e = \bot$. One of them is exactly the first line of Fig. 10. The other term is similar and can be evaluated efficiently in exactly the same way.

Comparison with FOVE The main algorithmic difference between FOVE [12] and the lifted inference mechanism of Bach is that the so-called shattering operation is performed up-front in FOVE but the equivalent operation is only performed dynamically on a per need basis in Bach. [12] rightly points out that the up-front approach can be expensive and unnecessary and lists as a future work an extension of their algorithm to do shattering dynamically. We believe what we have is close to, if not exactly, the extension wanted in [12].

4.3.3 Urn and Balls

We will see in this section how the points on the graph in Fig. 2 in Example 8 are computed. Expression (7) above can be written down formally as

$$\frac{1}{K} \times \sum_{s \in sb} \sum_{l \in (bd\ s)} (joint\ d\ (m, s, l, [o_1, \ldots, o_d])), \tag{36}$$

where $sb : \{\{Ball\}\}$

$sb = \lambda s.((setOfBalls\ m\ s) > 0)$

$bd : \{Ball\} \rightarrow \{(List\ Ball)\}$

$(bd\ s) = \lambda l.((ballsDrawn\ d\ s\ l) > 0).$

We have seen in Example 10 how the sets sb and $(bd\ s)$ can be enumerated by Bach. So we could just go ahead and compute the value of the term (36) as it is; but we would have to wait a long time – a very long time. This is because the cardinality of sb is 2^m, and that of $(bd\ s)$ is m^d. Both are exponential in the relevant parameters.

At this point, we can use Monte-Carlo integration techniques to estimate the probabilities we are interested in. (Indeed this has been done and reasonable estimates were obtained.) But a closer inspection of the problem reveals that even though the term (36) is expensive to evaluate, the actual complexity of the underlying computational problem is quite low. We will now show how term (36) can be simplified through a series of rewrites.

To begin with, we can obtain through the variable-elimination procedure the following equivalent but simpler expression for (36):

$$C \times \sum_{s \in sb} \sum_{l \in (bd\ s)} (observations\ l\ [o_1, \ldots, o_d]), \tag{37}$$

where $C = (numOfBalls\ m)0.5^m/(Km^d)$.

$$\sum_{s\in 2^{all}} \underline{\sum_{h\in 2^{othr}} (f\ (s\ Mike)\ \bot)(g\ (s\ Mike)\ \top)\Big(\prod_{x\in vst\setminus\{Mike\}} (f\ (s\ x)\ \top)(g\ (s\ x)\ \top)\Big)\Big(\prod_{x\in othr} (f\ (s\ x)\ (h\ x))(g\ (s\ x)\ \top)\Big)} \quad [24]$$

$$\underline{\sum_{s\in 2^{all}} (f\ (s\ Mike)\ \bot)(g\ (s\ Mike)\ \top)\Big(\prod_{x\in vst\setminus\{Mike\}} (f\ (s\ x)\ \top)(g\ (s\ x)\ \top)\Big)\sum_{h\in 2^{othr}}\prod_{x\in othr} (f\ (s\ x)\ (h\ x))(g\ (s\ x)\ \top)} \quad [33]$$

$$\Big(\sum_{s\in 2^{\{Mike\}}} (f\ (s\ Mike)\ \bot)(g\ (s\ Mike)\ \top)\Big)\underline{\sum_{s\in 2^{all\setminus\{Mike\}}}\Big(\prod_{x\in vst\setminus\{Mike\}} (f\ (s\ x)\ \top)(g\ (s\ x)\ \top)\Big)\sum_{h\in 2^{othr}}\prod_{x\in othr} (f\ (s\ x)\ (h\ x))(g\ (s\ x)\ \top)} \quad [33]$$

$$\Big(\sum_{s\in 2^{\{Mike\}}} (f\ (s\ Mike)\ \bot)(g\ (s\ Mike)\ \top)\Big)\Big(\underline{\sum_{s\in 2^{vst\setminus\{Mike\}}}\prod_{x\in vst\setminus\{Mike\}} (f\ (s\ x)\ \top)(g\ (s\ x)\ \top)}\Big)\sum_{s\in 2^{othr}}\sum_{h\in 2^{othr}}\prod_{x\in othr} (f\ (s\ x)\ (h\ x))(g\ (s\ x)\ \top) \quad [32]$$

$$\Big(\sum_{s\in 2^{\{Mike\}}} (f\ (s\ Mike)\ \bot)(g\ (s\ Mike)\ \top)\Big)\Big(\sum_{\pi\in M_2^{|vst\setminus\{Mike\}|}} C_\pi \prod_{x'\in\{\top,\bot\}} ((f\ x'\ \top)(g\ x'\ \top))^{(\mu_\pi\ x')}\Big)\underline{\sum_{s\in 2^{othr}}\sum_{h\in 2^{othr}}\prod_{x\in othr} (f\ (s\ x)\ (h\ x))(g\ (s\ x)\ \top)} \quad [31]$$

$$\Big(\sum_{s\in 2^{\{Mike\}}} (f\ (s\ Mike)\ \bot)(g\ (s\ Mike)\ \top)\Big)\Big(\sum_{\pi\in M_2^{|vst\setminus\{Mike\}|}} C_\pi \prod_{x'\in\{\top,\bot\}} ((f\ x'\ \top)(g\ x'\ \top))^{(\mu_\pi\ x')}\Big)\underline{\sum_{s\in 2^{othr}}\prod_{x\in othr}\sum_{h\in 2^{\{x\}}} (f\ (s\ x)\ (h\ x))(g\ (s\ x)\ \top)} \quad [31]$$

$$\underline{\Big(\sum_{s\in 2^{\{Mike\}}} (f\ (s\ Mike)\ \bot)(g\ (s\ Mike)\ \top)\Big)\Big(\sum_{\pi\in M_2^{|vst\setminus\{Mike\}|}} C_\pi \prod_{x'\in\{\top,\bot\}} ((f\ x'\ \top)(g\ x'\ \top))^{(\mu_\pi\ x')}\Big)}\prod_{x\in othr}\sum_{s\in 2^{\{x\}}}\sum_{h\in 2^{\{x\}}} (f\ (s\ x)\ (h\ x))(g\ (s\ x)\ \top)$$

$$\vdots$$

$$7.75\times\underline{\Big(\sum_{\pi\in M_2^{|vst\setminus\{Mike\}|}} C_\pi \prod_{x'\in\{\top,\bot\}} ((f\ x'\ \top)(g\ x'\ \top))^{(\mu_\pi\ x')}\Big)}\prod_{x\in othr}\sum_{s\in 2^{\{x\}}}\sum_{h\in 2^{\{x\}}} (f\ (s\ x)\ (h\ x))(g\ (s\ x)\ \top)$$

$$\vdots$$

$$7.75\times 381.08\times\underline{\prod_{x\in othr}\sum_{s\in 2^{\{x\}}}\sum_{h\in 2^{\{x\}}} (f\ (s\ x)\ (h\ x))(g\ (s\ x)\ \top)}$$

$$\vdots$$

$$7.75\times 381.08\times 225$$

Fig. 10 A computation illustrating lifted inference.

Exploiting symmetries, (37) can be further simplified to

$$C \times \sum_{s \in S'} \binom{m}{(n\, s)} \times \sum_{l \in (bd\ s)} (observations\ l\ [o_1, \ldots, o_d]), \tag{38}$$

where $S' \equiv \{\ \{\ (1, Blue), (2, Blue), (3, Blue), \ldots, (m, Blue)\ \},$
$\{\ (1, Green), (2, Blue), (3, Blue), \ldots, (m, Blue)\ \},$
$\{\ (1, Green), (2, Green), (3, Blue), \ldots, (m, Blue)\ \},$
$\vdots$
$\{\ (1, Green), (2, Green), (3, Green), \ldots, (m, Green)\ \}\ \},$

and $(n\, s)$ is the number of green balls in the set s. To understand (38), observe that for each s in S', there are $\binom{m}{(n\, s)}$ sets in sb that are equivalent (modulo relabelling of balls) to s with respect to the second summation term. Note the reduction in complexity of the outer summation term from 2^m to $m + 1$ achieved by this rewrite.

Proceeding in a similar vein, we can further simplify (38) to

$$C \times \sum_{s \in S'} \binom{m}{(n\, s)} \times \sum_{l \in L'} (n\, s)^{(p\, l)} \times (m - (n\, s))^{d - (p\, l)} \times (observations\ l\ [o_1, \ldots, o_d]), \tag{39}$$

where

$$L' \equiv \{\ l \mid \exists x_1 \cdots \exists x_d.((colour\ x_1) \wedge \cdots \wedge (colour\ x_d) \wedge l = [(1, x_1), \ldots, (d, x_d)])\ \},$$

and $(p\, l)$ is the number of green balls in the list l. To understand (39), observe that for each s in S' and each l in L', there are $(n\, s)^{(p\, l)}(m - (n\, s))^{d-(p\, l)}$ lists in the set $(bd\ s)$ that are equivalent to l with respect to the term $(observations\ l\ [o_1, \ldots, o_d])$. This last rewrite reduces the complexity of the inner summation from m^d to 2^d, a significant improvement.

But after all the above we still have a computational problem that is exponential in d. There is not much one can do to further simplify (39) in general. But we are dealing with a special case of (39) where $o_i = o_j$, for all $i, j \in \{1, \ldots, d\}$. In this special case, (39) can be further simplified to

$$C \times \sum_{s \in S'} \binom{m}{(n\, s)} \times \sum_{l \in L''} \binom{m}{(p\, l)} \times (n\, s)^{(p\, l)} \times (m - (n\, s))^{d - (p\, l)} \times (observations\ l\ [o_1, \ldots, o_d]), \tag{40}$$

where $L'' \equiv \{\ [(1, Blue), (2, Blue), (3, Blue), \ldots, (d, Blue)],$
$[(1, Green), (2, Blue), (3, Blue), \ldots, (d, Blue)],$
$[(1, Green), (2, Green), (3, Blue), \ldots, (d, Blue)],$
$\vdots$
$[(1, Green), (2, Green), (3, Green), \ldots, (d, Green)]\ \}.$

This simplification can be made because we have

$$(observations\ l_1\ [o, \ldots, o]) = (observations\ l_2\ [o, \ldots, o])$$

whenever l_1 and l_2 have the same number of green and blue balls; the order in which the colours appear does not actually matter. The computational complexity of evaluating expression (40) is a very manageable $O(|S'| \times |L''|) = O(m \times d)$, and this is what we used to compute the points in Fig. 2.

Comparison with BLOG From a modelling perspective, our solution to the urn and balls problem is very different to the solution presented in [40]. The Bayesian network that resulted from the BLOG problem specification has an infinite number of nodes. In contrast, our network has four, although some of the nodes are fairly complex in that they range over structured data like sets of balls and lists of balls.

We believe the compactness of our model provides better control when it comes to inference. Given a query, we can perform exact inference if that is feasible. We only need to resort to approximate-inference techniques when the answer to the query is inherently expensive to compute. This is different to BLOG. The default (and only) inference mechanism of BLOG is a sampling based method. The sheer size of the networks they generate makes sampling inevitable. Indeed, in BLOG, even simple queries that can be answered exactly cheaply must be handled approximately via sampling. As an example, consider the query discussed in this section. We have seen that the query can be answered exactly and cheaply. (Admittedly some non-trivial human input is provided here and a lot more needs to be automated in the future. This is addressed further in Section 6.) The probabilities calculated in Fig. 2 are all exact answers (at least up to the precision we require). But this same query can only be answered approximately using sample-based estimates in [40]. The main advantage of the BLOG scheme over the current Bach solution is that inference is completely automatic.

5 Learning

In this section, we discuss some learning problems and investigate one of these in some detail to see how it can be addressed using Bach.

5.1 Learning Problems

To understand the various kinds of learning problems that arise, it is helpful to consider the task of building agent systems – a setting that is sufficiently general to cover a large majority of computer applications. So consider an agent situated in some environment that can receive percepts from the environment and can apply actions that generally have a non-deterministic effect on the environment. The primary task of the agent is to do the 'right thing', that is, choose the appropriate action for each state it finds itself in, where 'appropriate' usually means maximising its expected performance measure. Associated with such a situation, there are several learning problems that we now explain.

Suppose that *State* is the type of states of the world, *Action* is the type of actions, and *Observation* is the type of observations that the agent can make with its sensors. Then *Density State* is the type of densities of states and *Density Observation* is the type of densities of observations. There are five important functions on these types that an agent must have available in order to choose actions. These are

$transition : Action \to State \to (Density\ State)$

$observe : State \rightarrow (Density\ Observation)$

$observationUpdate : Observation \rightarrow (Density\ State) \rightarrow (Density\ State)$

$policy : (Density\ State) \rightarrow Action$

$actionUpdate : Action \rightarrow (Density\ State) \rightarrow (Density\ State)$

Given an action and a state, the function *transition* returns a state density which gives the distribution on the states the agent could end up in as a result of applying the action to the current state. Generally, this function has to be learned during deployment of the agent by collecting training examples of the effect of applying actions.

The function *observe*, which provides the observation model, is generally given by the agent designer, although in more complex applications it may need to be learned from training examples.

The next function *observationUpdate* provides the update of the state density as a result of the agent perceiving a particular observation. Updating the state density is regarded as a learning task since the agent learns its new state density from the current state density and the information provided by the observation. Essentially, this update is an application of Bayes rule.

The function *policy*, the most important of the functions that the agent needs, gives the action that is appropriate for any particular state density. Assuming that the agent has learned the transition function and knows the utility of each state, this function can be defined directly. In cases where one or other of these assumptions is not true, some form of reinforcement learning may be needed to learn the policy.

Finally, the function *actionUpdate* provides the update of the state density as a result of the agent applying some action. This update is a simple computation using the transition function and the (current) state density, and is not usefully regarded as a learning task (except insofar as it requires knowledge of the transition function that usually needs to be learned).

In addition, many agent applications require another kind of learning task. In complex applications it is common for the state space to be very large indeed. In such cases, a common technique is to define features on the state space and work instead with the much smaller set of equivalence classes of states that are defined by the features. Sometimes features need to be learned during deployment of the agent using training examples [39].

In summary, many agent applications involve at least some of the four basic learning problems:

1. Learning the transition function.
2. Learning the policy function.
3. Learning features on the state space.
4. Updating the state density as a result of an observation.

The first three of these learning problems have much in common: in each case there are training examples that are obtained during deployment and there is an hypothesis space that is searched to obtain an hypothesis that 'explains' the training examples. More detail about these problems, in the context of using higher-order logic for knowledge representation, is given in [35,44,37].

The fourth problem is conceptually simpler than the first three in that it is handled by an application of Bayes rule but, depending on the application, there can be some subtle difficulties in this. We now show how updating state densities can be handled in robotics applications.

5.2 Updating State Densities

In a robotic system, the sensors can usually gather only partial, noisy, information about the state of the world. This noisy sensor data is integrated over time using Bayes rule to produce a distribution over possible states of the world. As this distribution represents the subjective belief of the agent about the state of the world it is referred to as the belief state.

In more detail, an agent is supplied with an initial (prior) distribution over states, a motion model (corresponding to *transition* above) defining a distribution over the resulting state in the world for each action performed from each starting state, and a sensor model (corresponding to *observe* above) which defines a distribution over the observations we make when in a given state. We introduce the following functions to represent the prior distribution and the motion and sensor models.

$$prior : (Density\ State)$$
$$motionModel : Action \to State \to (Density\ State)$$
$$sensorModel : State \to (Density\ Observation)$$

Here we have only given the type declarations. The actual definitions of these functions are of course problem-dependent and we will see some examples shortly. Assuming these functions are defined, we can use the following equations obtained using Bayes rule to update our state density when our robot performs an action or makes an observation about the world:

$$motionUpdate : Action \to (Density\ State) \to (Density\ State)$$
$$(motionUpdate\ a\ ds) = (ds \S (motionModel\ a)) \tag{41}$$

$$observationUpdate : Observation \to (Density\ State) \to (Density\ State)$$
$$(observationUpdate\ o\ ds) = (normalise\ \lambda s.((ds\ s) \times (sensorModel\ s\ o))), \tag{42}$$

where $normalise : (State \to Real) \to (Density\ State)$ normalises a real-valued function so that it sums/integrates to one over its domain. (The actual definition is dependent on the type *State*.)

As a simple specific example, imagine we have an iron-ore loader that runs on a track with four stations. Station 1 is under the conveyor belt from the stockpile, and the remaining stations are over different train lines. The ore loader must gather iron ore at Station 1 and then move over to one of the other stations and dump the iron ore into a carrying car.

The loader only has a simple sensor: it can sense whether it is under the conveyor with 90% accuracy. The loader can perform two actions: move to the next station or move to the previous one. These actions only work 85% of the time because wheels can slip. The loader fails to move at all 10% of the time. In the remaining 5% of the time, the loader moves two stations (in the desired direction). If the cart is limited by the length of the track then the distribution is modified accordingly. Using $1, 2, 3, 4$ to represent the states and Y, N to represent observations, we have the following formalisation:

$$(prior\ s) = 0.25$$
$$(sensorModel\ s\ o) = if\ ((s = 1) = (o = Y))\ then\ 0.9\ else\ 0.1$$

State Distribution				Action	Observation	Actual State
1	2	3	4			
0.25	0.25	0.25	0.25	-	-	2
0.0357	0.321	0.321	0.321	-	n	2
0.00357	0.0625	0.307	0.627	*next*	-	3
0.00040	0.0627	0.308	0.629	-	n	3
0.07223	0.2996	0.565	0.063	*prev*	-	2
0.00858	0.3202	0.604	0.067	-	n	2
0.32693	0.5488	0.117	0.007	*prev*	-	1
0.81383	0.1518	0.033	0.002	-	y	1

Table 1 The state distribution of the loader given a series of observations and actions.

$$
\begin{aligned}
&(\mathit{motionModel}\ a\ s\ s') = \\
&\quad (\mathit{if}\ (s = s')\ \mathit{then}\ 0.1\ \mathit{else\ if}\ (\mathit{distance}\ a\ s\ s') = 1\ \mathit{then}\ 0.85 \\
&\quad \mathit{else\ if}\ (\mathit{distance}\ a\ s\ s') = 2\ \mathit{then}\ 0.05\ \mathit{else}\ 0) \\
&+\ (\mathit{if}\ (a = \mathit{next}) \wedge (s' = 4)\ \mathit{then} \\
&\qquad (\mathit{if}\ (s = 4)\ \mathit{then}\ 0.9\ \mathit{else\ if}\ (s = 3)\ \mathit{then}\ 0.05\ \mathit{else}\ 0) \\
&\quad \mathit{else\ if}\ (a = \mathit{prev}) \wedge (s' = 1)\ \mathit{then} \\
&\qquad (\mathit{if}\ (s = 1)\ \mathit{then}\ 0.9\ \mathit{else\ if}\ (s = 2)\ \mathit{then}\ 0.05\ \mathit{else}\ 0) \\
&\quad \mathit{else}\ 0)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{distance} : \mathit{Action} \to \mathit{State} \to \mathit{State} \to \mathit{Int} \\
&(\mathit{distance}\ a\ s\ s') = \mathit{if}\ (a = \mathit{next})\ \mathit{then}\ (s' - s)\ \mathit{else}\ (s - s').
\end{aligned}
$$

Leaving aside the problem of control, the location of the loader can now be tracked. Table 1 shows the state distribution after two observations and an action.

In practice most robotics problems are too large to use a simple discrete set of states as we did in the previous example. Luckily, there are some known distributions where the two update rules (41) and (42) can be performed in closed form. These so-called *conjugate priors* can be defined in the system and allow one to track the state of real-world problems.

For example, we can define a k-dimensional Gaussian distribution that takes a mean vector and a covariance matrix as arguments as follows:

$$
\begin{aligned}
&\mathit{gaussian} : \mathit{Vector} \to \mathit{Matrix} \to (\mathit{Density}\ \mathit{Vector}) \\
&(\mathit{gaussian}\ \mu\ \Sigma) = \lambda x. \frac{1}{\sqrt{(2\pi)^k |\Sigma|}} e^{-\frac{1}{2}(x-\mu)^T \Sigma^{-1} (x-\mu)}.
\end{aligned}
$$

Using identities such as

$$
\begin{aligned}
&(\mathit{normalise}\ \lambda s.((\mathit{gaussian}\ \mu_1\ \Sigma_1)\ s) \times ((\mathit{gaussian}\ \mu_2\ \Sigma_2)\ s)) = \\
&\qquad (\lambda n.(\mathit{gaussian}\ \ (\Sigma_1 n \mu_2 + \Sigma_2 n \mu_1)\ \ (\Sigma_1 n \Sigma_2))\ \ (\Sigma_1 + \Sigma_2)^{-1}), \qquad (43)
\end{aligned}
$$

the system can rewrite the update rules defined above in Equations (41) and (42) in closed form. This allows us to track the state of a k-dimensional continuous linear system – this is mathematically equivalent to a Kalman filter [28]. For example, imagine we have a robot moving about a two-dimensional world. The robot's state will be represented as a two-dimensional vector. The robot can move; its actions will also

Actual Location		Desired Action		Actual Action		Observed Location		Estimated State		
x	y	x	y	x	y	x	y	mean		Variance
0.0	0.0	5.0	0.0	4.881	0.035			5.0	0.0	1.1
4.881	0.035					5.227	-0.228	5.208	-0.209	0.092
4.881	0.035	5.0	5.0	5.053	4.721			10.208	4.791	0.192
9.934	4.756					10.136	4.717	10.161	4.743	0.066
9.934	4.756	0.0	5.0	0.009	5.110			10.161	9.743	0.166
9.943	9.866					9.806	9.929	9.940	9.859	0.062
9.943	9.866	-5.0	0.0	-4.909	0.744			4.940	9.860	0.162
5.034	10.610					5.477	10.827	5.272	10.458	0.062
5.034	10.610	-5.0	-5.0	-4.788	-5.628			0.272	5.458	0.162
0.246	4.982					0.256	4.744	0.262	5.017	0.062
0.246	4.982	0.0	-5.0	0.050	-5.006			0.262	0.017	0.162
0.296	-0.024					0.143	-0.543	0.189	-0.329	0.062

Table 2 The actual and estimated state of a 2D robot given a series of noisy actions and observations. In this example, the covariance matrix is always diagonal with equal entries, and so for compactness we have only given one number for the variance.

be represented by a two-dimensional vector describing the relative motion from the current location. When the robot moves, there is circular Gaussian noise with variance 0.1 in its movement. The robot also has an approximate location sensor which returns the robot's location with additional circular Gaussian noise of variance 0.1. The above leads to the following theory:

$$(prior\ s) = (gaussian\ [0.0, 0.0]\ [[1.0, 0.0], [0.0, 1.0]]\ s)$$

$$(sensorModel\ s\ o) = (gaussian\ o\ [[0.1, 0.0], [0.0, 0.1]]\ s)$$

$$(motionModel\ a\ s\ s') = (gaussian\ (s + a)\ [[0.1, 0.0], [0.0, 0.1]]\ s').$$

Consider an agent that starts at the origin, but with uncertainty about its own location represented by a circular Gaussian distribution of variance 1. The agent then attempts to perform a series of actions which, if performed correctly, would bring it back to its starting location. As noted above, there is noise in these actions. The agent will track its location using Bayesian tracking. The path travelled by the agent on one simulated run and an estimate of its state at each step as calculated by Bach is shown in Table 2.

6 Discussion

This section contains some historical background, more discussion about higher-order logic, a comparison with other probabilistic reasoning systems, and several other topics.

Integrating Logic and Probability The problem of integrating logic and probability has a history going back around 300 years for which three main threads can be discerned. The oldest by far is the philosophical thread that can be traced via Boole [4,5] back to Jacob Bernoulli in 1713. An extensive historical account of this thread can be found in [23] and overviews of more recent work in [24,58]. The second thread is that of the knowledge representation and reasoning community in artificial intelligence, of which [46,25,18,26,55] are typical works. The third thread is that of the machine learning community in artificial intelligence, of which [42,9,40,52,41,29] are typical works.

An important and useful technical distinction that can be made between these various approaches is that the combination of logic and probability can be done externally or internally [59]: in the external view, probabilities are attached to formulas in some logic; in the internal view, formulas incorporate statements about probability. One can even mix the two cases so that probabilities appear both internally and externally. This paper takes the the internal view. By way of comparison, we now briefly discuss the external view.

The standard logical setting adopted for integrating logic and probability according to the external view is first-order logic. Imagine that an agent is operating in some environment for which there is some uncertainty (for example, the environment might be partially observable). The environment is modelled as a probability distribution over the collection of first-order interpretations (over some suitable alphabet for the application at hand). The intuition is that any of these interpretations could be the actual environment but that some interpretations are more likely than others and this information is given by the distribution on the interpretations. If the agent actually knew this distribution, then it could answer probabilistic questions of the form: if (closed) formula ψ holds, what is the probability that the (closed) formula φ holds? In symbols, the question is: what is $Pr(\varphi \,|\, \psi)$?

We now formalise this situation. Let $\mathfrak{I}$ be the set of interpretations and p a probability measure on the σ-algebra of all subsets of this set. Define the random variable $X_\varphi : \mathfrak{I} \to \mathbb{R}$ by

$$X_\varphi(I) = \begin{cases} 1 & \text{if } \varphi \text{ is true in } I \\ 0 & \text{otherwise,} \end{cases}$$

with a similar definition for X_ψ. Then $Pr(\varphi \,|\, \psi)$ can be written in the form

$$p(X_\varphi = 1 \,|\, X_\psi = 1)$$

which is equal to

$$\frac{p(X_\varphi = 1 \wedge X_\psi = 1)}{p(X_\psi = 1)}$$

and, knowing p, can be evaluated.

Of course, the real problem is to know the distribution on the interpretations. To make some progress on this, most systems intending to integrate logical and probabilistic reasoning make simplifying assumptions. For a start, most are based on Prolog. Thus theories are first-order Horn clause theories, maybe with negation as failure. Interpretations are limited to Herbrand interpretations and often function symbols are excluded so the Herbrand base (and therefore the number of Herbrand interpretations) is finite. Let $\mathfrak{I}$ denote the (finite) set of Herbrand interpretations and $\mathcal{B}$ the Herbrand base. We can identify $\mathfrak{I}$ with the product space $\{0,1\}^{\mathcal{B}}$ in the natural way.[2] Thus the problem amounts to knowing the distribution on this product space. At this point, there is a wide divergence in the approaches. For example, the product distribution can be represented either directly or more compactly using Bayesian networks or Markov random fields. In [52], the occurrences of atoms in the same clause are used

[2] The Herbrand base $\mathcal{B}$ of a first-order language L is the set of all ground atoms in L. Each subset of $\mathcal{B}$ is a Herbrand interpretation. Each Herbrand interpretation I corresponds to an element in $\{0,1\}^{\mathcal{B}}$ in the sense that an atom in $\mathcal{B}$ is true (takes value 1) iff the atom is in I.

to give the arcs and the weights attached to clauses are used to give the potential functions in a Markov random field. In [29], conditional probability distributions are attached to clauses to give a Bayesian network. Closely related to [29] is [49], in which probability distributions are attached to sets of literals capturing alternative scenarios and logic programming is used to generate a distribution on possible worlds. In [53], Prolog is extended with probabilistic switches to define distributions over Herbrand interpretations. In [3], A-Prolog [20] is extended with probabilistic atoms to generate distributions on possible worlds using answer set programming. In all these cases, the logic is exploited to give some kind of compact representation of what is usually a very large distribution. Generally, the theory is only used to define/construct/constrain the underlying distribution on possible models and reasoning proceeds probabilistically as described above, either through specially developed proof procedures or (modified versions of) standard inference algorithms for graphical models. To achieve efficiency in answering queries, most systems employ some kind of knowledge-based model construction technique that constructs only those parts of the underlying distribution that are relevant to the query.

There are some interesting contrasts between the above external view and what we propose in this paper. Here we adopt the standard axiomatic method of using a theory to model a situation and relying on the soundness of theorem proving to produce results that are correct in the intended interpretation. We simply have to note that this theory, if it is higher-order, can include densities that can be reasoned with. In our approach, whatever the situation, there is a single intended interpretation, which would include densities in the case where uncertainty is being modelled, that is a model of the theory. Our approach also gives fine control over exactly what uncertainty is modelled – we only introduce densities in those parts of the theory that really need them. Furthermore, the probabilistic and non-probabilistic parts of a theory work harmoniously together.

Another attractive aspect of our approach is that the correctness of even sophisticated probabilistic-inference procedures can be obtained easily as a consequence of the soundness theorem for the general theorem-proving procedure. A case in point is lifted inference. The two algorithms proposed in [50] and [12] are quite complex and establishing their correctness is non-trivial. What we show in this paper is that the essential operations of those two algorithms can be distilled down to several equations in Bach, and these when acted on by the equational-reasoning mechanism of Bach give us the desired lifted-inference procedure. The correctness of our lifted-inference procedure then follows easily from Theorem 1 and the correctness of the individual equations.

Why Higher-order Logic? The most notable technical attribute of the material in this paper is that it is set in the context of higher-order logic. Since almost all other work on the topic of this paper is given in the context of first-order logic, we make some comments about this more general setting. In summary, our view is that the higher-order setting is superior to the first-order setting. To justify this claim, we now examine the two settings from several points of view. For a highly readable account of this topic that is more detailed than we have space for here, we strongly recommend [19].

The first aspect is that of expressive power. Higher order-logic, also known as simple type theory [6], is highly expressive. One way to think about higher-order logic is that it is a formalisation of everyday informal mathematics. Mathematical concepts are easy to express directly in higher-order logic because, amongst other things, the logic allows quantification over predicates and functions. This is illustrated by the direct modelling of probabilistic concepts such as densities and operations on them in

higher-order theories; other good examples are given in [19]. In contrast, first-order logic only allows one to model many mathematical concepts indirectly and requires the introduction of (semantically complicated) set theory to give a satisfactory foundation for mathematics. The great expressive power of higher-order logic partly explains its widespread use in some subfields of computer science; in functional programming, where a program can be understood as a higher-order equational theory; in formal methods, where the logic is used to give specifications of programs and prove properties about them; in theoretical computer science, where various kinds of semantics are typically higher order; and elsewhere. This expressivity part of the story is quite convincing: it is *much* easier to express many mathematical concepts, function definitions in functional programming languages, and program specifications, as well as probabilistic concepts like those discussed in this paper, in higher-order rather than first-order logic. (How they might be expressed at all in first-order logic is discussed below.)

However, even accepting the superior expressive power of higher-order logic, a common criticism is that it is computationally less attractive than first-order logic. This criticism is usually fuelled by observations such as the fact that higher-order unification is undecidable [14] and the logic does not have a sound and complete proof system. Carefully formulated, these criticisms are correct, but they do not present a balanced view of the situation. For that, we need to say something about the semantics of higher-order logic.

In the semantics, each (closed) type α is interpreted by a (non-empty) set D_α. The crucial aspect of the semantics in this discussion is the meaning given to function types. In the standard semantics, for a function type $\alpha \to \beta$, the set $D_{\alpha\to\beta}$ is *all* functions from D_α to D_β. The models given by this semantics are called standard models. Gödel showed in 1931 that, with the standard semantics, higher-order logic does not have a sound and complete proof system. Also (higher-order versions of) the compactness theorem and the Löwenhein-Skolem theorem do not hold.

The characteristic of the standard semantics that leads to these undesirable properties is that there are comparatively few models. Before we explain how the problem can be fixed, note that a desirable aspect of this semantics is that many theories are *categorical*, that is, have exactly one model up to isomorphism. For example, the theory of a complete ordered field has just the real numbers as a model (up to isomorphism). This is exactly what one would want: the theory has characterised just the desired model. Also there is a proof system for the logic that, while not complete, is an adequate foundation for mathematics. For a discussion of this, see [19].

A way to get a completeness result is to expand the class of models. This was famously done by Henkin in 1950 [27]. The key idea is to expand the class of models by allowing $\alpha \to \beta$ to denote a *subset* of the set of all functions from D_α to D_β, not necessarily all functions. The class of models given by this definition are called general models. Each standard model is a general model, but the converse is not true. With this enlarged set of models, Henkin was able to prove that there is a sound and complete proof procedure for higher-order logic. Also, using the Henkin semantics, the compactness theorem and the Löwenheim-Skolem theorem hold.

Note that, at this point, Lindström's (first) theorem [15] can be applied to show that higher-order logic with the Henkin semantics is essentially just a variant of first-order logic. (Informally, Lindström's theorem states that a logical system (satisfying some weak technical conditions) that is at least as strong as first-order logic and satisfies conditions corresponding to the compactness theorem and Löwenheim-Skolem theorem is equally strong as first-order logic; here, 'at least as strong' and 'equally strong' are

technical, model-theoretic notions.) In spite of this result, something important has been gained: instead of being forced to express certain concepts awkwardly in first-order logic, the greater expressive power of higher-order logic can be exploited.

It is also interesting to see how a theory T in higher-order logic can be directly encoded as a theory T' in first-order logic. Since there is a simple embedding of many-sorted first-order logic into (unsorted) first-order logic [16], it suffices to embed higher-order logic into many-sorted logic. In outline, this is done as follows. (More detail is given in [19].) For each (closed) type α in T, there is a sort s_α in the many-sorted theory T'. Each variable and constant of type α in T is represented by a variable and constant, respectively, of sort s_α in T'. For each function type $\alpha \to \beta$ of higher-order logic, there is an ('apply') function in T' having sort $s_{\alpha\to\beta} \times s_\alpha \to s_\beta$ that represents the application of functions of type $\alpha \to \beta$ to arguments of type α. For each abstraction in T, there is a function in T' that represents the abstraction. The theory T' also includes extensionality and comprehension axioms. As a result of this encoding, a general model of T is represented by a many-sorted model of T'. Overall, the encoding shows that a theory of higher-order logic with the Henkin semantics is essentially just a first-order theory presented in a more convenient form. But, to emphasise yet again, something important has been gained by working in the higher-order context: the higher-order syntax is natural for expressing concepts whose encoding into first-order logic would be unnatural.

With regard to the undecidability of higher-order unification, note first that we cannot avoid dealing with undecidability even in the first-order case, since the validity problem of first-order logic is undecidable. In any case, one can do a lot in higher-order logic without ever having to resort to (higher-order) unification. This should be evident from the computational models of widely used functional languages like Haskell and ML, all of which are highly efficient and effective. Our reasoning system Bach, which can be viewed as an extension of Haskell, is also a useful subset of higher-order logic that is both expressive and tractable. It uses linear-time (one-way) matching of terms instead of the difficult (two-way) unification of terms for pattern matching. Also it captures a significant part of theorem-proving via a computationally inexpensive mechanism for doing equational reasoning. The general strategy adopted here is of course no different from the common technique of restricting first-order logic in different ways to achieve tractability in inference.

In summary, with the standard semantics, higher-order logic admits the desirable categorical theories and there is a proof system adequate for the foundations of mathematics, but the price one pays is the lack of a complete proof system. With the Henkin semantics, one loses categoricity, but gains important properties that first-order logic has such as a sound and complete proof system, the compactness theorem, and the Löwenhein-Skolem theorem, and yet has available the more expressive higher-order syntax.

Other Systems based on Higher-order Languages We move on now to a discussion of related systems based on higher-order logic, starting with a comparison of Bach and IBAL [48], the system closest to ours in the literature. IBAL has three components: a probabilistic-reasoning component, a learning component, and a decision-making component. We focus only on the probabilistic-reasoning component here.

IBAL and Bach are in some ways quite similar. Each is based on a higher-order language, and each represents probability densities using that language. However, there

are significant differences between the two in the details of how a distribution is represented and the computational mechanisms used to answer queries.

To represent a distribution, IBAL extends a standard functional programming language with stochastic features. Terms in the IBAL language represent a distribution over values, rather than a single value as in a standard programming language. These terms are still used in the language as if they contained a single value and IBAL takes care of calculating the appropriate distribution over the result. In this way, an IBAL program describes a generative model for a joint distribution. This generative model can be constrained with *observation* statements. These statements allow the results of a generative model to be partially specified, and those specifications used to inform the values of random variables appearing earlier in the model. An *observation* statement in IBAL converts a generative model into a conditional model. IBAL's method of representing a distribution is quite different from that used in Bach, where probability distributions are represented explicitly as functions and manipulated as such.

From a language design perspective, there is also a philosophical difference between the two systems in that Bach allows the description of distributions it may not be able reason with efficiently, whereas the IBAL language is restricted to fit its computational mechanism. In fact, any language features that would be inefficient to reason with using IBAL's inference mechanism were removed.

Once a model has been specified, IBAL uses specific probabilistic inference routines to answer queries about the model. The exact computational model appears to have changed in different versions of the IBAL system. [48] describes a two stage process. A graphical representation of the computation is generated in the first stage. This representation is then processed to answer the query in the second stage. The intermediate graphical structure is lazily generated and is closely related to the function call graph, a much smaller structure than the fully expanded graphical model.

As we have seen, IBAL's computational model is specialised for probabilistic inference. In contrast, Bach uses a general computational model for higher-order logic. We then have various equations represented in our language that allow the system to reason efficiently with many common density functions. As it is not possible, or desirable, to list all equations that might speed up computation, we also allow the user to add their own tailored rewrites within the system. Some of these equations (e.g. (43)) require Bach's programming with abstractions facility [33,35], which allows matching and reduction inside lambda expressions. This facility is not available in more standard functional computation models like those underlying ML, Haskell, and IBAL.

We could gain significantly from the addition of some syntactic sugar to ease common use cases. Having said this, the flexibility of the Bach language, and the ability for the user to specify useful equations, allow a user of our system to easily extend its capabilities. For example, in Section 5.2, we show how Bach can be extended to efficiently reason with a continuous Gaussian distribution even though our system cannot efficiently reason with continuous distributions in general.

We now give a survey of other related work. The system described in [22] is based on the same underlying logic as ours. In it, the authors explore ways of defining probability distribution functions over basic terms [35], a class of terms in higher-order logic identified for the purpose of representing individuals in applications. All the distributions in the class identified in [22] are defined by induction on the structure of basic terms. An efficient sampling algorithm is also given for these distributions. The work set up in [22] fits directly into our general framework.

There are also related studies on adding probabilistic reasoning support to functional programming languages. These come in the form of language extensions to the λ-calculus. In [51], which is closely related to IBAL, the authors show how probability distributions can be captured using monads. They also introduced a simple language called measure terms that allows certain operations on (discrete) distributions to be computed efficiently. Measure terms is a subset of the language we use for representing and manipulating densities in this paper. In [47], a probabilistic language based on sampling functions is presented. The language supports all kinds of probability distributions and exploits the fact that sampling functions form a so-called state monad. In [21], a system similar to IBAL is introduced that uses sampling as its main inference mechanism.

In [17], the authors build on the idea that probability distributions form a monad to design a probability and simulation library for Haskell. In [1], the Haskell language is used to describe a family of statistical models based on the MML principle. The generalisation of the various models, and the natural mappings between them, are shown by the use of Haskell classes, types, and functions.

Programming versus Intelligent Knowledge Base To what extent are we *just* implementing different probabilistic models and inference procedures in Bach? In other words, is Bach a programming approach or an intelligent knowledge base approach?

The first thing to note is that the (pure) programming approach and the (fully automatic) intelligent knowledge base approach are just two ends of a spectrum of possibilities. So the question to ask is not whether Bach is a programming approach or an intelligent knowledge base approach, but where in the spectrum does it lie. For the purpose of this discussion, the spectrum of possibilities can be characterised along two dimensions: expressive power and the level of automation in inference. (There are other dimensions like efficiency as well, of course.) Expressive power here refers to the availability of language features and the general compactness of model descriptions (with respect to our target class of probabilistic AI applications). In particular, we are not talking about notions like Turing completeness where there is no difference between, for example, assembly language and Prolog. Fig. 11 shows where we see the different languages sit in this scheme of things.

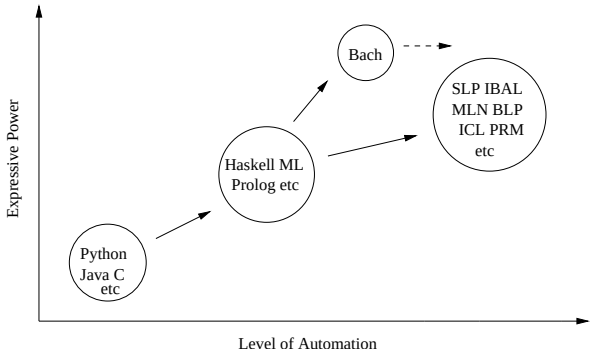


Fig. 11 A mapping of the different languages

Naturally, the top right-hand corner of the graph is where we want to be. Low-level imperative languages like C and Java are quite clearly not what we want unless speed is the biggest consideration. Widely used declarative programming languages like Haskell and Prolog are better but these languages are designed with full programming generality in mind and they can be significantly improved for our target area of probabilistic AI applications, both in terms of expressive power and the level of automated inference supported. However, there is a general trade-off to be made here since increased expressive power is usually accompanied by a harder automated-inference problem. The difference between Bach and many existing SRL systems can be understood in terms of this tradeoff. By design, Bach leans towards greater expressive power while many existing SRL systems lean towards more sophisticated level of automated inference.

At present, Bach allows the user to specify, in a declarative fashion, a wide class of probabilistic models. We have seen several different probabilistic models in this paper. In [43], we also show how the different forms of graphical models, probabilistic logic programs [45,31,13] and representative first-order probabilistic logics like [18] can be captured in Bach. The class of probabilistic models we can accommodate is larger than that supported in any one existing system. For this generality, we pay a price in the form of a more difficult inference problem. There remains work to be done on automating more of Bach's probabilistic inference mechanism. In posing queries to Bach, the user needs to know how to marginalise densities and enumerate sets. This is not too onerous since the tools needed for both these operations are well supported in the language. In any case, it would not be hard to introduce syntactic sugar for common probabilistic queries that can be mechanically translated into terms involving the required marginalisation operations. The harder problem for Bach is in automatically figuring out ways to efficiently compute (approximate) answers to queries posed. As we saw in the urn and balls example (§4.3.3), naively posing the correct question is not always sufficient to obtain the desired answer in a reasonable amount of time. In this case, a bit of cleverness on the part of the user solves the problem. What is required in general is a mechanism to automatically detect difficult integration/summation operations and then solve them using Monte-Carlo integration/summation techniques. This is one issue that we are working on in the on-going development of Bach.

Coming back to our original question posed in the opening paragraph, the answer is that Bach sits somewhere in between a programming approach and an intelligent knowledge base approach, but is moving in the direction of the latter.

Approximate Inference A good probabilistic logical language needs to support approximate inference for the case when computing the exact answer is unacceptably expensive. Support for approximate inference in Bach is still under development. When describing a probabilistic system in our language, there are a number of approaches. Our preferred approach is that the system is described naturally in logic, without the user requiring any special knowledge of probabilistic inference. When input in this natural form there are many distributions that can be described, but about which Bach, with only support for a general purpose equational reasoner, cannot efficiently answer questions exactly. It would be desirable to incorporate an approximation system into Bach to handle such cases. One of the main components needed for such a system is an efficient way to sample from terms. Towards that end, we have developed a scheme whereby suitable sampling distributions for a wide range of inference tasks can be specified and efficiently sampled from. This work is ongoing and will be reported elsewhere.

7 Conclusion

We conclude by summarising the main contributions of the paper.

1. A detailed account of a theory-based approach to probabilistic modelling, inference, and learning is presented.
2. Higher-order logic is shown to be an expressive and practical logic in which to model and reason about applications involving uncertainty, and thus provide an harmonious integration of logic and probability.
3. A detailed account of the theory and application of lifted probabilistic inference, which clarifies the existing literature on this topic, is presented.
4. It is shown how Bach can be used to support probabilistic modelling and inference, for both the discrete and continuous cases, in a wide range of applications.

Acknowledgments

We thank Peter Cheeseman, Marcus Hutter, Kristian Kersting, Brian Milch, Scott Sanner, Tim Sears, Joel Veness, and Yang Wang for helpful discussions. We also thank the anonymous reviewers whose comments helped improve this paper. NICTA is funded by the Australian Government's Department of Communications, Information Technology, and the Arts and the Australian Research Council through Backing Australia's Ability and the ICT Research Centre of Excellence programs.

References

1. Lloyd Allison. Models for machine learning and data mining in functional programming. *Journal of Functional Programming*, 15(1):15–32, 2005.
2. Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Academic Press, 1986.
3. Chitta Baral, Michael Gelfond, and J. Nelson Rushton. Probabilistic reasoning with answer sets. In *Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 21–33, 2004.
4. George Boole. *An Investigation of the Laws of Thought on which are founded the Mathematical Theories of Logic and Probabilities*. Walton and Maberly, 1854.
5. George Boole. *Studies in Logic and Probability*. Watts & Co, 1952.
6. Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.
7. Keith Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, 1978.
8. Joshua J. Cole, Matthew Gray, John W. Lloyd, and Kee Siong Ng. Personalisation for user agents. In F. Dignum et al, editor, *Proceedings of the 4th International Joint Conference on Autonomous Agents and Multi Agent Systems*, pages 603–610, 2005.
9. Luc De Raedt and Kristian Kersting. Probabilistic logic learning. *SIGKDD Explorations*, 5(1):31–48, 2003.
10. Rodrigo de Salvo Braz. *Lifted First-Order Probabilistic Inference*. PhD thesis, University of Illinois at Urbana-Champaign, 2007.
11. Rodrigo de Salvo Braz, Eyal Amir, and Dan Roth. Lifted first-order probabilistic inference. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, pages 1319–1325, 2005.
12. Rodrigo de Salvo Braz, Eyal Amir, and Dan Roth. Lifted first-order probabilistic inference. In Lise Getoor and Ben Taskar, editors, *Introduction to Statistical Relational Learning*, chapter 15. MIT Press, 2007.
13. Alex Dekhtyar and V. S. Subrahmanian. Hybrid probabilistic programs. *Journal of Logic Programming*, 43(3):187–250, 2000.

14. Gilles Dowek. Higher-order unification and matching. In *Handbook of automated reasoning*, pages 1009–1062. Elsevier Science Publishers B. V., 2001.
15. H.D. Ebbinghaus, J. Flum, and W. Thomas. *Mathematical Logic*. Springer-Verlag, 1984.
16. Herbert B. Enderton. *A Mathematical Introduction to Logic*. Harcourt Academic Press, 2nd edition, 2001.
17. Martin Erwig and Steve Kollmansberger. Probabilistic functional programming in Haskell. *Journal of Functional Programming*, 16(1):21–34, 2006.
18. Ronald Fagin and Joseph Y. Halpern. Reasoning about knowledge and probability. *Journal of the ACM*, 41(2):340–367, 1994.
19. William M. Farmer. The seven virtues of simple type theory. *Journal of Applied Logic*, 6(3):267–286, 2008.
20. Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–387, 1991.
21. Noah Goodman, Vikash K. Mansinghka, Daniel Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: A language for generative models. In David A. McAllester and Petri Myllymäki, editors, *Proceedings of the 24th Conference in Uncertainty in Artificial Intelligence*, pages 220–229. AUAI Press, 2008.
22. Elias Gyftodimos and Peter A. Flach. Combining Bayesian networks with higher-order data representations. In *6th International Symposium on Intelligent Data Analysis*, pages 145–156, 2005.
23. T. Hailperin. *Sentential Probability Logic*. Lehigh University Press, 1996.
24. Alan Hájek. Probability, logic and probability logic. In L. Goble, editor, *The Blackwell Guide to Philosophical Logic*, chapter 16, pages 362–384. Blackwell, 2001.
25. Joseph Y. Halpern. An analysis of first-order logics of probability. *Artificial Intelligence*, 46(3):311–350, 1990.
26. Joseph Y. Halpern. *Reasoning about Uncertainty*. MIT Press, 2003.
27. Leon Henkin. Completeness in the theory of types. *Journal of Symbolic Logic*, 15(2):81–91, 1950.
28. R.E. Kalman. A new approach to linear filtering and prediction problems. *Transactions of the ASME: Journal of Basic Engineering*, 82(D):35–45, 1960.
29. Kristian Kersting and Luc De Raedt. Bayesian logic programming: Theory and tool. In Lise Getoor and Ben Taskar, editors, *Introduction to Statistical Relational Learning*, chapter 10. MIT Press, 2007.
30. Frank R. Kschischang, Brendan J. Frey, and Hans-Andrea Loeliger. Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory*, 47(2), 2001.
31. Laks V. S. Lakshmanan and Fereidoon Sadri. Modeling uncertainty in deductive databases. In D. Karagiannis, editor, *Proceedings of the International Conference on Database and Expert Systems Applications, DEXA'94*, pages 724–733, 1994.
32. Daniel Leivant. Higher-order logic. In D.M. Gabbay, C.J. Hogger, J.A. Robinson, and J. Siekmann, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 2, pages 230–321. Oxford University Press, 1994.
33. John W. Lloyd. Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming*, 3, 1999.
34. John W. Lloyd. Knowledge representation, computation, and learning in higher-order logic. Available at `http://rsise.anu.edu.au/~jwl/`, 2002.
35. John W. Lloyd. *Logic for Learning: Learning Comprehensible Theories from Structured Data*. Springer, 2003.
36. John W. Lloyd. Knowledge representation and reasoning in modal higher-order logic. Available at `http://rsise.anu.edu.au/~jwl`, 2007.
37. John W. Lloyd and Kee Siong Ng. Learning modal theories. In S. Muggleton, R. Otero, and A. Tamaddoni-Nezhad, editors, *Proceedings of the 16th International Conference on Inductive Logic Programming*, LNAI 4455, pages 320–334, 2007.
38. John W. Lloyd, Kee Siong Ng, and Joel Veness. Modal functional logic programming. Available at `http://rsise.anu.edu.au/~kee`, 2007.
39. John W. Lloyd and Tim D. Sears. An architecture for rational agents. In *Declarative Agent Languages and Technologies (DALT-2005)*, LNAI 3904, pages 51–71. Springer, 2006.
40. Brian Milch, Bhaskara Marthi, Stuart Russell, David Sontag, Daniel L. Ong, and Andrey Kolobov. BLOG: Probabilistic models with unknown objects. In L.P. Kaelbling and A. Saffiotti, editors, *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, pages 1352–1359, 2005.

41. Brian Milch and Stuart Russell. First-order probabilistic languages: Into the unknown. In S. Muggleton, R. Otero, and A. Tamaddoni-Nezhad, editors, *Proceedings of the 16th International Conference on Inductive Logic Programming*, LNAI 4455, pages 10–24, 2007.
42. Stephen Muggleton. Stochastic logic programs. In L. De Raedt, editor, *Advances in Inductive Logic Programming*, pages 254–264. IOS Press, 1996.
43. Kee Siong Ng and John W. Lloyd. Probabilistic reasoning in a classical logic. *Journal of Applied Logic*, 2008. doi:10.1016/j.jal.2007.11.008.
44. K.S. Ng. *Learning Comprehensible Theories from Structured Data*. PhD thesis, Computer Sciences Laboratory, The Australian National University, 2005.
45. Raymond T. Ng and V. S. Subrahmanian. Probabilistic logic programming. *Information and Computation*, 101(2):150–201, 1992.
46. Nils J. Nilsson. Probabilistic logic. *Artificial Intelligence*, 28(1):71–88, 1986.
47. Sungwoo Park, Frank Pfenning, and Sebastian Thrun. A probabilistic language based upon sampling functions. In *Conference Record of the 32nd Annual ACM Symposium on Principles of Programming Languages*, 2005.
48. Avi Pfeffer. The design and implementation of IBAL: A general-purpose probabilistic language. In Lise Getoor and Ben Taskar, editors, *Introduction to Statistical Relational Learning*, chapter 14. MIT Press, 2007.
49. David Poole. Logic, knowledge representation, and Bayesian decision theory. In *Proceedings of the 1st International Conference on Computational Logic*, LNCS 1861, pages 70–86, 2000.
50. David Poole. First-order probabilistic inference. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, pages 985–991, 2003.
51. Norman Ramsey and Avi Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *Conference Record of the 29th Annual ACM Symposium on Principles of Programming Languages*, 2002.
52. Matthew Richardson and Pedro Domingos. Markov logic networks. *Machine Learning*, 62:107–136, 2006.
53. Taisuke Sato and Yoshitaka Kameya. PRISM: A language for symbolic-statistical modeling. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, pages 1330–1335, 1997.
54. Stewart Shapiro. Classical logic II – Higher-order logic. In L. Goble, editor, *The Blackwell Guide to Philosophical Logic*, pages 33–54. Blackwell, 2001.
55. A. Shirazi and E. Amir. Probabilistic modal logic. In R.C. Holte and A. Howe, editors, *Proceedings of the 22nd AAAI Conference on Artificial Intelligence*, pages 489–495, 2007.
56. Simon Thompson. *Type Theory and Functional Programming*. Addison-Wesley, 1991.
57. Johan van Benthem and Kees Doets. Higher-order logic. In D.M. Gabbay and F. Guenther, editors, *Handbook of Philosophical Logic*, volume 1, pages 275–330. Reidel, 1983.
58. Jon Williamson. Probability logic. In D. Gabbay, R. Johnson, H.J. Ohlbach, and J. Woods, editors, *Handbook of the Logic of Inference and Argument: The Turn Toward the Practical*, volume 1 of *Studies in Logic and Practical Reasoning*, pages 397–424. Elsevier, 2002.
59. Jon Williamson. Philosophies of probability. In A. Irvine, editor, *Handbook of the Philosophy of Mathematics, Volume 4 of the Handbook of the Philosophy of Science*. Elsevier, 2008. In press.
60. Nevin Lianwen Zhang and David Poole. A simple approach to Bayesian network computations. In *Proceedings of the 10th Biennial Canadian Artificial Intelligence Conference*, pages 171–178, 1994.

A Lifted Inference Proofs

A.1 Proof of Inversion Elimination

Proposition 1 *Let t be a real-valued term such that each free occurrence of the variable f in t occurs in a subterm of the form $(f\ x_1\ \ldots\ x_k)$, where $x_1, \ldots, x_k$ are free occurrences in t of these variables and $1 \leq k \leq n$, and $X_1, \ldots, X_n, Y$ be non-empty finite sets. Then*

$$\sum_{f \in Y^{X_n}} \prod_{x_1 \in X_1} \cdots \prod_{x_k \in X_k} t = \prod_{x_1 \in X_1} \cdots \prod_{x_k \in X_k} \sum_{f \in Y^{X_n^{\{x_k\}}}} t.$$

Proof The proof is by induction on k.

Base case. We have to show that

$$\sum_{f \in Y^X} \prod_{x \in X} t = \prod_{x \in X} \sum_{f \in Y^{\{x\}}} t.$$

(Y can itself be a function space.) Suppose that $X = \{x_1, \ldots, x_n\}$, $Y = \{y_1, \ldots, y_m\}$, and $Y^X = \{f_1, \ldots, f_{m^n}\}$. Define $f_{ij} \in Y^{\{x_i\}}$ by $f_{ij}(x_i) = y_j$, for $i = 1, \ldots, n$ and $j = 1, \ldots, m$. Then

$$\begin{aligned}
&\sum_{f \in Y^X} \prod_{x \in X} t \\
&= \sum_{f \in Y^X} t\{x/x_1\} \times t\{x/x_2\} \times \cdots \times t\{x/x_n\} \\
&= \quad t\{x/x_1, f/f_1\} \times t\{x/x_2, f/f_1\} \times \cdots \times t\{x/x_n, f/f_1\} \\
&\quad + t\{x/x_1, f/f_2\} \times t\{x/x_2, f/f_2\} \times \cdots \times t\{x/x_n, f/f_2\} \\
&\quad + \cdots \\
&\quad + t\{x/x_1, f/f_{m^n}\} \times t\{x/x_2, f/f_{m^n}\} \times \cdots \times t\{x/x_n, f/f_{m^n}\} \\
&= \quad (t\{x/x_1, f/f_{11}\} + t\{x/x_1, f/f_{12}\} + \cdots + t\{x/x_1, f/f_{1m}\}) \\
&\quad \times (t\{x/x_2, f/f_{21}\} + t\{x/x_2, f/f_{22}\} + \cdots + t\{x/x_2, f/f_{2m}\}) \\
&\quad \times \cdots \\
&\quad \times (t\{x/x_n, f/f_{n1}\} + t\{x/x_n, f/f_{n2}\} + \cdots + t\{x/x_n, f/f_{nm}\}) \quad \text{[Assumption on } t\text{]} \\
&= \prod_{x \in X} \sum_{f \in Y^{\{x\}}} t.
\end{aligned}$$

Induction step.

$$\begin{aligned}
&\sum_{f \in Y^{X_n}} \prod_{x_1 \in X_1} \cdots \prod_{x_k \in X_k} t \\
&= \prod_{x_1 \in X_1} \sum_{f \in Y^{X_n^{\{x_1\}}}} \prod_{x_2 \in X_2} \cdots \prod_{x_k \in X_k} t \quad \text{[Base case]} \\
&= \prod_{x_1 \in X_1} \sum_{g \in Y^{X_n}} \prod_{x_2 \in X_2} \cdots \prod_{x_k \in X_k} t\{(f\ x_1)/g\} \\
&= \prod_{x_1 \in X_1} \prod_{x_2 \in X_2} \cdots \prod_{x_k \in X_k} \sum_{g \in Y^{X_n^{\{x_k\}}}} t\{(f\ x_1)/g\} \quad \text{[Induction hypothesis]}
\end{aligned}$$

$$= \prod_{x_1 \in X_1} \cdots \prod_{x_k \in X_k} \sum_{f \in Y^{X_n}_{k+1}^{\cdot\cdot^{\cdot\{x_k\}}}} t^{\cdot\cdot^{\cdot\{x_1\}}}$$

A.2 Proof of Counting Elimination

Let M^n_m, C_π and μ_π be as defined in Section 4.2.3. If X is a set of cardinality n, Y is a set of cardinality m, $\phi : Y \to \{1, \ldots, m\}$ a fixed, but arbitrary, bijection, and $\pi \equiv (k_1, \ldots, k_m) \in M^n_m$, then

$$Y^X|_\pi = \{f \in Y^X \mid \phi(y) = i \text{ implies } |f^{-1}(y)| = k_i, \text{ for each } y \in Y\}.$$

Note that the cardinality of $Y^X|_\pi$ is C_π.

Proposition 2 *Let t be a real-valued term such that each free occurrence of the variable f_i in t occurs in a subterm of the form $(f_i\ x_{i1})$ or ... or $(f_i\ x_{ir_i})$, where $x_{i1}, \ldots, x_{ir_i}$ are free occurrences in t of these variables that do not have any free occurrences in t other than these, for $i = 1, \ldots, p$. Let $y_{i1}, \ldots, y_{ir_i}$ be variables not appearing in t, for $i = 1, \ldots, p$. Also let X_i be a set of cardinality $n_i \geq 1$ and Y_i a set of cardinality $m_i \geq 1$, for $i = 1, \ldots, p$. Then*

$$\sum_{f_1 \in {Y_1}^{X_1}} \cdots \sum_{f_p \in {Y_p}^{X_p}} \prod_{x_{11} \in X_1} \cdots \prod_{x_{1r_1} \in X_1} \cdots \prod_{x_{p1} \in X_p} \cdots \prod_{x_{pr_p} \in X_p} t$$

$$= \sum_{\pi_1 \in M^{n_1}_{m_1}} \cdots \sum_{\pi_p \in M^{n_p}_{m_p}} \prod_{i=1}^{p} C_{\pi_i} \prod_{y_{11} \in Y_1} \cdots \prod_{y_{1r_1} \in Y_1} \cdots \prod_{y_{p1} \in Y_p} \cdots \prod_{y_{pr_p} \in Y_p}$$

$$t\{(f_1\ x_{11})/y_{11}, \ldots, (f_p\ x_{pr_p})/y_{pr_p}\}^{\mu_{\pi_1}(y_{11}) \cdot \ldots \cdot \mu_{\pi_1}(y_{1r_1}) \cdot \ldots \cdot \mu_{\pi_p}(y_{p1}) \cdot \ldots \cdot \mu_{\pi_p}(y_{pr_p})}.$$

Proof

$$\sum_{f_1 \in {Y_1}^{X_1}} \cdots \sum_{f_p \in {Y_p}^{X_p}} \prod_{x_{11} \in X_1} \cdots \prod_{x_{1r_1} \in X_1} \cdots \prod_{x_{p1} \in X_p} \cdots \prod_{x_{pr_p} \in X_p} t$$

$$= \sum_{\pi_1 \in M^{n_1}_{m_1}} \cdots \sum_{\pi_p \in M^{n_p}_{m_p}} \sum_{f_1 \in {Y_1}^{X_1}|_{\pi_1}} \cdots \sum_{f_p \in {Y_p}^{X_p}|_{\pi_p}} \prod_{x_{11} \in X_1} \cdots \prod_{x_{1r_1} \in X_1} \cdots \prod_{x_{p1} \in X_p} \cdots \prod_{x_{pr_p} \in X_p} t$$

$$= \sum_{\pi_1 \in M^{n_1}_{m_1}} \cdots \sum_{\pi_p \in M^{n_p}_{m_p}} \sum_{f_1 \in {Y_1}^{X_1}|_{\pi_1}} \cdots \sum_{f_p \in {Y_p}^{X_p}|_{\pi_p}} \prod_{y_{11} \in Y_1} \cdots \prod_{y_{1r_1} \in Y_1} \cdots \prod_{y_{p1} \in Y_p} \cdots \prod_{y_{pr_p} \in Y_p}$$

$$t\{(f_1\ x_{11})/y_{11}, \ldots, (f_p\ x_{pr_p})/y_{pr_p}\}^{\mu_{\pi_1}(y_{11}) \cdot \ldots \cdot \mu_{\pi_1}(y_{1r_1}) \cdot \ldots \cdot \mu_{\pi_p}(y_{p1}) \cdot \ldots \cdot \mu_{\pi_p}(y_{pr_p})}$$

$$= \sum_{\pi_1 \in M^{n_1}_{m_1}} \cdots \sum_{\pi_p \in M^{n_p}_{m_p}} \prod_{i=1}^{p} C_{\pi_i} \prod_{y_{11} \in Y_1} \cdots \prod_{y_{1r_1} \in Y_1} \cdots \prod_{y_{p1} \in Y_p} \cdots \prod_{y_{pr_p} \in Y_p}$$

$$t\{(f_1\ x_{11})/y_{11}, \ldots, (f_p\ x_{pr_p})/y_{pr_p}\}^{\mu_{\pi_1}(y_{11}) \cdot \ldots \cdot \mu_{\pi_1}(y_{1r_1}) \cdot \ldots \cdot \mu_{\pi_p}(y_{p1}) \cdot \ldots \cdot \mu_{\pi_p}(y_{pr_p})}.$$

The step from the second to the third expression uses the condition on t. □