The Design and Implementation of IBAL: A General-Purpose Probabilistic Language

Avi Pfeffer

TR-12-05

Computer Science Group
Harvard University
Cambridge, Massachusetts

The Design and Implementation of IBAL: A General-Purpose Probabilistic Language

Avi Pfeffer AVI@EECS.HARVARD.EDU
Harvard University,
Division of Engineering and Applied Science
33 Oxford St,
Cambridge, MA 02138 USA

Abstract

This paper describes IBAL, a high level representation language for probabilistic AI. IBAL integrates several aspects of probability-based rational behavior, including probabilistic reasoning, Bayesian parameter estimation and decision theoretic utility maximization. IBAL is based on the functional programming paradigm, and is an ideal rapid prototyping language for probabilistic modeling. The paper presents the IBAL language, and presents a number of examples in the language. It then discusses the semantics of IBAL, presenting the semantics in two different ways. Finally, the inference algorithm of IBAL is presented. Seven desiderata are listed for inference, and it is shown how the algorithm fulfils each of them.

1. Introduction

In a rational programming language, a program specifes a situation encountered by an agent; evaluating the program amounts to computing what a rational agent would believe or do in the situation. Rational programming combines the advantages of declarative representations with features of programming languages such as modularity, compositionality, and type systems. A system designer need not reinvent the algorithms for deciding what the system should do in each possible situation it encounters. It is sufficient to declaratively describe the situation, and leave the sophisticated inference algorithms to the implementors of the language.

One can think of Prolog as a rational programming language, focused on computing the beliefs of an agent that uses logical deduction. In the past few years there has been a shift in AI towards specifications of rational behavior in terms of probability and decision theory. There is therefore a need for a natural, expressive, general-purpose and easy to program language for probabilistic modeling. This paper presents IBAL, a probabilistic rational programming language. IBAL, pronounced "eyeball", stands for Integrated Bayesian Agent Language. As its name suggests, it integrates various aspects of probability-based rational behavior, including probabilistic reasoning, Bayesian parameter estimation and decision-theoretic utility maximization.

High-level probabilistic languages have generally fallen into two categories. The first category is rule-based (Poole, 1993; Ngo & Haddawy, 1996; Kersting & de Raedt, 2000). In this approach the general idea is to associate logic-programming-like rules with noise factors. A rule describes how one first-order term depends on other terms. Given a specific

query and a set of observations, a Bayesian network can be constructed describing a joint distribution over all the first-order variables in the domain.

The second category of language is object-based (Koller & Pfeffer, 1997, 1998; Laskey & Mahoney, 1997). In this approach, the world is described in terms of objects and the relationships between them. Objects have attributes, and the probabilistic model describes how the attributes of an object depend on other attributes of the same object and on attributes of related objects. The model specifies a joint probability distribution over the attributes of all objects in the domain.

This paper explores a different approach to designing high-level probabilistic languages. IBAL is a functional language for specifying probabilistic models. Models in IBAL look like programs in a functional programming language. In the functional approach, a model is a description of a computational process. The process stochastically generates a value, and the meaning of the model is the distribution over the value generated by the process.

The functional approach, as embodied in IBAL, has a number of attractive features. First of all, it is an extremely natural way to describe a probabilistic model. To construct a model, one simply has to provide a description of the way the world works. Describing the generative process explicitly is the most direct way to describe a generative model. Second, IBAL is highly expressive. It builds on top of a Turing-complete programming language, so that every generative model that can reasonably be described computationally can be described in IBAL. Third, by basing probabilistic modeling languages on programming languages, we are able to enjoy the benefits of a programming language, such as a type system and type inference. Furthermore, by building on the technology of functional languages, we are able to utilize all their features, such as lambda abstraction and higher-order functions.

In addition, the use of a functional programming framework provides an elegant and uniform language with which to describe all aspects of a model. All levels of a model can be described in the language, including the low-level probabilistic dependencies and the high-level structure. This is in contrast to rule-based approaches, in which combination rules describe how the different rules fit together. It is also in contrast to object-based languages, in which the low-level structure is represented using conditional probability tables and a different language is used for high-level structure. Furthermore, PRMs use special syntax to handle uncertainty over the relational structure. This means that each such feature must be treated as a special case, with special purpose inference algorithms. In IBAL, special features are encoded using the language syntax, and the general-purpose inference algorithm is applied to handle them.

IBAL is an ideal rapid prototyping language for developing new probabilistic models. Several examples are provided that show how easy it is to express models in the language. These include well-known models as well as new models. IBAL has been implemented, and made publicly available at `http:www.eecs.harvard.edu/~avi/IBAL`.

The paper begins by presenting the IBAL language. The initial focus is on the features that allow description of generative probabilistic models. In addition, IBAL allows observations to be included explicitly in models. IBAL also contains constructs to support learning of probabilistic parameters in a program. There are also constructs for describing decision problems.

After presenting examples, the paper presents the semantics of IBAL. Two different approaches to defining the semantics are given. The first defines a sampling experiment,

such that the meaning of a program is the distribution over outputs of the experiment. The second defines the semantics directly in terms of probability distributions over values. The use of lazy semantics, in which values are only generated as needed, is also discussed.

When implementing a highly expressive reasoning language, the question of inference comes to the forefront. Because IBAL is capable of expressing many different frameworks, its inference algorithm should generalize the algorithms of those frameworks. If, for example, a Bayesian network is encoded in IBAL, the IBAL inference algorithm should perform the same operations as a Bayesian network inference algorithm. This paper describes the IBAL inference algorithm and shows how it generalizes many existing frameworks, including Bayesian networks, hidden Markov models, and stochastic context free grammars. Seven desiderata for a general-purpose inference algorithm are presented, and it is shown how IBAL's algorithm satisfies all of them simultaneously.

2. The IBAL Language

IBAL is a rich language. We describe it in stages. First we describe the basic expressions which are used to build generative probabilistic models. We then discuss how to encode observations in models, how to describe parameter estimation problems, and how to define utilities and decision problems. Finally we describe some syntactic sugar that makes the language easier and more natural to use.

2.1 Basic Expressions

The basic program unit in IBAL is the *expression*. An expression describes a stochastic experiment that generates a value. Just as in a regular programming language an expression describes a computation that produces a value, so in IBAL an expression describes a computation that stochastically produces a value. IBAL provides constructs for defining basic expressions, and for composing expressions together to produce more complex expressions. In this section we provide an intuitive meaning for IBAL expressions in terms of stochastic experiments. We will provide precise semantics in Section 4. The core of IBAL includes the following kinds of expressions.

Constant expressions A constant expression is a literal of one of the built-in primitive types, Boolean, Integer and Symbol. The Symbol type contains symbolic constants, which can be any string value. For example, `true`, 6 and `'hello` are all constant expressions. A constant expression represents the experiment that always produces the given value.

Conditional expressions The expression `if` e_1 `then` e_2 `else` e_3 provides conditional choice between two possible outcomes. It corresponds to the experiment in which e_1 is evaluated; then, if the value of e_1 was `true`, e_2 is evaluated, otherwise e_3 is evaluated.

Stochastic choice The expression `dist [` p_1 `:` e_1`, ...,` p_n `:` e_n `]` specifies a stochastic choice among the different possibilities $e_1, \ldots, e_n$. Each of the p_i is the probability of choosing the corresponding e_i. The expression corresponds to the experiment in which the i-th branch is chosen with probability p_i, and then the expression e_i is evaluated.

Variable binding IBAL allows variables to be named and assigned a value, and then referred to later. This can be accomplished using an expression of the form `let` x `=` e_1 `in` e_2. Here x is the name of the variable being defined, e_1 is its definition, and e_2 is the expression in which x can appear. The simplest way to understand a `let` expression is that it corresponds to the experiment in which e_1 is evaluated, and then e_2 is evaluated, with the result of e_1 being used wherever x appears. The result of the entire `let` expression is the result of e_2.

Lambda abstraction IBAL provides lambda abstraction, allowing the definition of functions. The expression `lambda` $x_1, \ldots, x_n$ `->` e represents the function that takes arguments $x_1, \ldots, x_n$ whose body is e. Function definitions can also be recursive, using the syntax `fix` $f(x_1, \ldots, x_n)$ `->` e. Here f is the name of the function being defined, and the body e can refer to f. Both `lambda` and `fix` expressions correspond to experiments that always produce the functional value defined by the expression. The functional value is a closure consisting of argument names, a function body, and an environment in which to evaluate free variables.

Function application The expression $e_0(e_1, \ldots, e_n)$ represents function application. It corresponds to the experiment in which e_0 is evaluated, and its functional result is applied to the results of $e_1, \ldots, e_n$. Note that there may be uncertainty in e_0, the expression defining the function to be applied.

Tuple construction and access The expression $< x_1 : e_1, \ldots, x_n : e_n >$ constructs a tuple with components named $x_1, \ldots, x_n$. It corresponds to the experiment in which each of the e_i is evaluated and assigned to the component x_i. Once a tuple has been constructed, a component can be accessed using dot notation. The expression $e.x$ evaluates the expression e, and extracts component x from the result.

Example 2.1: It is important to note that in an expression of the form `let` x `=` e_1 `in` e_2 the variable x is assigned a specific value in the experiment; any stochastic choices made while evaluating e_1 are resolved, and the result is assigned to x. For example, consider

```
let z = dist [ 0.5 : true, 0.5 : false ] in
z & z
```

The value of `z` is resolved to be either `true` or `false`, and the same value is used in the two places in which `z` appears in `z & z`. Thus the whole expression evaluates to `true` with probability 0.5, not 0.25 which is what the result would be if `z` was reevaluated each time it appears. Thus the `let` construct provides a way to make different parts of an expression probabilistically dependent, by making them both mention the same variable. ∎

Example 2.2: This example illustrates the use of a higher order function. It begins by defining two functions, one corresponding to the toss of a fair coin and one describing a toss of a biased coin. It then defines a higher-order function, whose return value is one of the first two functions. This corresponds to the act of deciding which kind of coin to toss. The example then defines a variable named `c` whose value is either the `fair` or `biased` function. It then defines two variables `x` and `y` to be different applications of the function contained

in c. The variables x and y are conditionally independent of each other given the value of c. Note by the way that in this example the functions take zero arguments.

```
let fair = fun () -> dist [ 0.5 : 'heads, 0.5 : 'tails ] in
let biased = fun () -> dist [ 0.9 : 'heads, 0.1 : 'tails ] in
let pick = fun () -> dist [ 0.5 : fair, 0.5 : biased ] in
let c = pick () in
let x = c () in
let y = c () in
<x:x, y:y>
```

■

2.2 Observations

The previous section presented the basic constructs for describing generative probabilistic models. Using the constructs above, one can describe any stochastic experiment that generatively produces values. The language presented so far can express many common models, such as Bayesian networks, probabilistic relational models, hidden Markov models, dynamic Bayesian networks and stochastic context free grammars. All these models are generative in nature. The richness of the model is encoded in the way the values are generated.

IBAL also provides the ability to describe conditional models, in which the generative probability distribution is conditioned on certain observations being satisfied. IBAL achieves this by allowing observations to be encoded explicitly in a model, at any point. An observation serves to condition the model on the observation being true.

An observation has the general syntax `obs` $x = v$ `in` e where x is a variable, v is a value and e is an expression. Its meaning is the same as expression e, except that the value of variable x is conditioned to be equal to v. The variable x should have been defined earlier, as part of a `let` expression.

Example 2.3: Consider

```
let y = dist [ 0.5 : true, 0.5 : false ] in
let z =
  if y
  then dist [ 0.9 : true, 0.1 : false ]
  else dist [ 0.1 : true, 0.9 : false ] in
obs z = true in
y
```

Here, the distribution defined by the expression is the conditional distribution over y, given that z takes on the value true. ■

Observations can appear anywhere, including within function definitions. Ordinarily, a function represents a conditional probability distribution over outputs of the function given inputs. When an observation appears in the definition of a function it serves to modify this conditional distribution, to be conditioned on the observation being true. A function

is like a black box. It defines a conditional distribution. How the distribution is defined is irrelevant to any code that calls the function. The distribution may be defined purely generatively, or it may be defined through conditioning, using observations. If observations are used, they are fully encapsulated in the definition of the function.

2.3 Learnable Parameters

Observations provide the basis for integrating learning, in the form of Bayesian parameter estimation, into the IBAL framework. Unknown probability parameters are specified using *prior* expressions, which have the form `param` $x = [\alpha_1, \ldots, \alpha_n]$ `in` e. A prior expression defines a probabilistic parameter $\theta^x = \theta^x_1, \ldots, \theta^x_n$, and specifies a Dirichlet prior over the parameter. The α_i are positive real numbers, specifying the the hyperparameters of the Dirichlet. After the parameter has been declared, the expression e is evaluated.

The probabilistic parameter is used in the resulting expression e by using a `pdist` expression. A `pdist` expression is similar to a `dist` expression except that instead of using hard-coded parameters for the probabilities, a learnable parameter is used. The syntax of a `pdist` expression is `pdist` x $[e_1, \ldots, e_n]$. The meaning of a `pdist` expression is to choose the i-th subexpression with probability equal to the value of θ^x_i.

Example 2.4: Let us refine the coins example from earlier by adding priors and observations.

```
let fair = fun () -> dist [ 0.5 : 'heads, 0.5 : 'tails ] in
param pbiased = [ 90, 10 ] in
let biased = fun () -> pdist pbiased [ 'heads, 'tails ] in
param ppick = [ 1, 1 ] in
let pick = fun () -> pdist ppick [ fair, biased ] in
let c = pick () in
let x = c () in
let y = c () in
obs x = 'heads in
y
```

A fair coin is known to produce `'heads` with probability 0.5. The probability of `'heads` for a biased coin is unknown, but its prior is peaked around 0.9, while the prior over which coin gets picked is uniform. As before, `c` is the result of picking a coin, and `x` and `y` are two tosses of `c`. We also have an observation that `x` came out `'heads`. This observation has multiple effects. First, because `'heads` is more likely for a biased coin, the probability that `c` is biased is increased, which in turn increases the probability that `y` is `'heads`. The observation also conditions the probability parameters. Because `c`, a result of applying `pick`, is likely to have turned out biased, we will get a posterior over the `ppick` parameter that is more weighted towards a biased result. Furthermore, because `c` may have been biased, and because a toss of `c` came out `'heads`, the posterior for the `pbiased` parameter is also weighed slightly more strongly towards `'heads`. ∎

With its learning component, IBAL is able to do parameter estimation for many common models, including hidden Markov models, stochastic context free grammars and probabilistic relational models. Furthermore, learning in IBAL is not just "added on" to the

probabilistic representation language, but is thoroughly integrated into the language. As a result, the benefits of compositionality and modularity are obtained for representing learning tasks. In particular, IBAL is good at representing a cumulative learning framework, in which smaller models are learned and then used as components of larger learning problems. Just as observations only condition values within their scope, they are only used to learn about model parameters within their scope. Thus a compositional learning process can be specified by providing a nested scope containing all the data and parameters for a learning subproblem, and a containing scope that uses the results of the subproblem.

2.4 Decisions and Utilities

In addition to describing probabilistic models, IBAL can represent decision problems in which an agent gets to make a decision and receives a reward. The representation of decision problems in IBAL is geared towards two popular models: influence diagrams (IDs) and Markov decision processes (MDPs). IBAL can easily represent these and other models, including various kinds of structured MDPs.

A decision declaration in IBAL has the form `choose` x `from` $s_1, \ldots, s_n$ `given` $x_1, \ldots, x_m$. This specifies the name x of the decision variable, its range $s_1, \ldots, s_n$, and the information available to the decision maker, the variables $x_1, \ldots, x_m$ (called the *informational parents* of x).

A reward expression is simply `reward` r `in` e. This specifies that a reward of r is received, and then the expression e is evaluated. One can also define the result of an expression to be the discounted reward of its subexpression, using the syntax `discount` d `in` e. The value of this expression is the value of e, except that the reward accumulated when evaluating e is discounted by the discount factor d.

Once `reward` expressions have been introduced, every stochastic experiment produces a reward as well as a value. In a `choose` expression, the value of the decision variable in a stochastic experiment will be the one that maximizes the expected reward of the result expression, given the previously generated values of the informational parents.

Example 2.5: We can adapt our coins example so that it includes a decision problem. The decision maker has to guess whether the coin `y` came out `heads` or `tails`, after observing `x`.

```
let fair = fun () -> dist [ 0.5 : 'heads, 0.5 : 'tails ] in
let biased = fun () -> dist [ 0.9 : 'heads, 0.1 : 'tails ] in
let pick = fun () -> dist [ 0.5 : fair, 0.5 : biased ] in
let c = pick () in
let x = c () in
let y = c () in
choose guess from 'heads, 'tails given x in
if guess == y
then reward 1 in 'correct
else reward -1 in 'incorrect
```

2.5 Syntactic Sugar

In addition to the basic constructs described above, IBAL provides a good deal of syntactic sugar. The sugar does not increase the expressive power of the language, but makes it considerably easier to work with. The syntactic sugar is presented here, because it will be used in many of the later examples.

Thus far, every IBAL construct has been an expression. Indeed, everything in IBAL can be written as an expression, and presenting everything as expressions simplifies the presentation. A real IBAL program, however, also contains *declarations*. A *block* is a piece of IBAL code consisting of a sequence of declarations. A declaration could be a variable definition, an observation, declaration of a learnable parameter, a reward, or a decision problem. Essentially, any language construct that appears with the keyword `in` can be turned into a declaration. Use of declarations modularizes code and makes it more readable.

Example 2.6: For example, we can rewrite our coins example using declarations.

```
fair = fun () -> dist [ 0.5 : 'heads, 0.5 : 'tails ]
params pbiased = [ 90, 10 ]
       ppick = [ 1, 1 ]
biased = fun () -> pdist pbiased [ 'heads, 'tails ]
pick = fun () -> pdist ppick [ fair, biased ]
c = pick ()
x = c ()
y = c ()
obs x = 'heads
```

The value of this block is a tuple containing a component for every variable defined in the block, i.e. `fair`, `biased`, `pick`, `c`, `x` and `y`. ∎

Bernoulli and uniform random variables are so common that a special notation is created for them. The expression `flip` α is shorthand for `dist` $[\alpha$: `true`$, 1-\alpha$: `false`$]$. The expression `uniform` n is short for `dist` $[\frac{1}{n} : 0, \ldots, \frac{1}{n} : n-1]$.

IBAL provides basic operators for working with values. These include logical operators for working with Boolean values and arithmetic operators for integer values. IBAL also provides an equality operator that tests any two values for equality. Operator notation is equivalent to function application, where the relevant functions are built in.

Dot notation can be used to reference nested components of variables. For example, `x.a.b` means the component named `b` of the component named `a` of the variable named `x`. This notation can appear anywhere a variable appears. For example, in an observation one can say `obs x.a = 'true in y`. This is equivalent to saying
`let z = x.a in obs z = 'true in y`.

Patterns can be used to match sets of values. A pattern may be

- an atomic value (Boolean, integer or strong), that matches itself;
- the special pattern *, that matches any value;

- a variable, which matches any value, binding the variable to the matched value in the process;
- a tuple of patterns, which matches any tuple value such that each component pattern matches the corresponding component value.

For example, the pattern $< 2, *, \texttt{y} >$ matches value $< 2, \texttt{true}, \texttt{'h} >$, binding `y` to `'h` in the process. A pattern can appear in an observation. For example, `obs x = <2,*,y> in 'true` conditions the experiment on the value of `x` matching the pattern.

Patterns also appear in `case` expressions, which allow the computation to branch depending on the value of a variable. The general syntax of `case` expressions is

$$\begin{array}{l}\texttt{case}\ e_0\ \texttt{of}\\ \#p_1 : e_1\\ \ldots\\ \#p_n : e_n\end{array}$$

where the p_i are patterns and the e_i are expressions. The meaning, in terms of a stochastic experiment, is to begin by evaluating e_0. Then its value is matched to each of the patterns in turn. If the value matches p_1, the result of the experiment is the result of e_1. If the value does not match p_1 through p_{i-1} and it does match p_i, then e_i is the result. It is an error for the value not to match any pattern. A `case` expression can be rewritten as a series of nested `if` expressions.

The `case` expression is useful for describing conditional probability tables as are used in Bayesian networks. In this case, the expression e_0 is a tuple consisting of the parents of the node, each of the patterns p_i matches a specific set of values of the parents, and the corresponding expression e_i is the conditional distribution over the node given the values of the parents. It is also possible to define a pattern that matches whenever a subset of the variables takes on specified values, regardless of the values of other variables. Such a pattern can be used to define conditional probability tables with context-specific independence, where only some of the parents are relevant in certain circumstances, depending on the values of other parents.

In addition to tuples, IBAL provides algebraic data types (ADTs) for creating structured data. An ADT is a data type with several variants. Each variant has a tag and a set of fields. ADTs are very useful in defining recursive data types such as lists and trees. For example, the list type has two variants. The first is `Nil` and has no fields. The second is `Cons` and has a field representing the head of the list and a further field representing the remainder of the list.

Example 2.7: Using the list type, we can easily define a stochastic context free grammar. First we define the `append` function that appends two lists. Then, for each non-terminal in the grammar we define a function corresponding to the act of generating a string with that non-terminal. For example,

```
append(x,y) =
  case x of
  # Nil -> y
```

```
  # Cons(a,z) -> Cons(a, append(z,y))
term(x) = Cons(x,Nil)
s() = dist [0.6:term('a);
            0.4:append(s(),t())]
t() = dist [0.9:term('b);
            0.1:append(t(),s())]
```

We can then examine the beginning of a string generated by the grammar using the `take` function:

```
take(n,x) =
  case(n,x) of
  # (0,_) -> Nil
  # (_,Nil) -> Nil
  # (_,Cons(y,z)) -> Cons(y,take(n-1,z))
```

The `let` syntax is extended to make it easy to define functions. The syntax `let` $f(x_1, \ldots, x_n) = e$ is equivalent to `let` $f =$ `fix` $f(x_1, \ldots, x_n) = e$.

In defining rewards, it is useful to be able to select the appropriate reward based on the state of the world. The `reward case` syntax is provided for this. The syntax is

`reward case` e_0 `of`
$\#p_1 : reward_1$
$\ldots$
$\#p_n : reward_n$
`in` e

This is similar to a `case` expression, in that the expression e_0 is evaluated and then matched against each of the patterns $p_1, \ldots, p_n$ in turn. Then, if the value of e_0 first matches p_i, a reward of r_i is received. The experiment then continues by evaluating the result expression e.

IBAL is a strongly typed language. The language includes `type` declarations that declare new types, and `data` declarations that define algebraic data types. The type language will not be presented here, but it will be used in the examples, where it will be explained.

In some cases, it is useful to define a condition as being erroneous. For example, when one tries to take the head of an empty list, an error condition should result. IBAL provides an expression `error` s, where s is a string, to signal an error condition. This expression takes on the special value `ERROR:` s, which belongs to every type and can only be used to indicate errors.

Finally, IBAL allows comments in programs. A comment is anything beginning with a // through to the end of the line.

3. Examples

Example 3.1:

Encoding a Bayesian network is easy and natural in IBAL. We include a definition for each variable in the network. A `case` statement is used to encode the conditional probability table for a variable. For example,

```
burglary = flip 0.01;

earthquake = flip 0.001;

alarm = case <burglary, earthquake> of
   # <false, false> : flip 0.01
   # <false, true> : flip 0.1
   # <true, false> : flip 0.7
   # <true, true> : flip 0.8
```

We can also easily encode conditional probability tables with structure. For example, we may want the `alarm` variable to have a noisy-or structure:

```
alarm = flip 0.01 // leak probability
  | earthquake & flip 0.1
  | alarm & flip 0.7
```

We may also create variables with context-specific independence. Context-specific independence is the case where a variable depends on a parent for some values of the other parents but not other. For example, if we introduce variables representing whether or not John is at home and John calls, John calling is dependent on the alarm only in the case that John is at home. IBAL's pattern syntax is very convenient for capturing context-specific independence. The symbol "*" is used as the pattern that matches all values, when we don't care about the value of a specific variable:

```
john_home = flip 0.5

john_calls = case <john_home, alarm> of
  # <false,*> : false
  # <true,false> : flip 0.001
  # <true,true> : flip 0.7
```

■

Example 3.2: Probabilistic relational models (Koller & Pfeffer, 1998) help bring structure to large scale Bayesian network models. They talk about the world in terms of the objects in it and the relationships between them. A model consists of a set of interconnected objects. Each object belongs to a class. A class has an associated probabilistic model. Objects have attributes, and the probabilistic model of a class specifies what the parents and conditional probability table are of each attribute. A PRM model describes the classes of object, and then specifies which instances of each class exist, and how they are related to each other.

A PRM can be specified in IBAL. The notion of class of object corresponds naturally to a function in IBAL. Each function contains definitions for each of the attributes of the

class represented by the function. The value of an instance of a class is a tuple containing values for all the attributes. In the following IBAL program, note how some instances (`o_chem` and `basket_weaving`) have their values specified explicitly, while other instances (`field1`, `prof1`, `prof2`, `course1`, `course2` and `student1`) are defined to be the results of applying their class. The distribution over values of these instances is as specified in the class model. The fact that `course1` is defined as `course(prof1, o_chem)` means that `course1` is related to `prof1` and `o_chem`. The `course` function represents the relationship that relates a professor and a field to a particular course object.

```
field() = { hard = flip 0.5; high_standards = flip 0.3 };

o_chem = { hard = true; high_standards = true };

basket_weaving = { hard = false; high_standards = false };

prof(field) = {
  mean = flip 0.1;
  tough = mean | (field.high_standards & flip 0.8);
  clear = ~ mean & (if field.hard then flip 0.5 else flip 0.8);
  field = field
};

course(prof, field) = {
  well_taught = (prof.clear | ~ field.hard) & flip 0.9;
  prof = prof;
};

student() = {
  smart = flip 0.4;
  hard_working = flip 0.7;
  good_test_taker = if smart then flip 0.8 else flip 0.3
};

performance(student, course) = {
  understands =
    if student.hard_working
    then case <student.smart, course.well_taught> of
    # <true, true> : flip 0.95
    # <false, false> : flip 0.35
    # _ : flip 0.7
    else case student.smart of
    # true : flip 0.7
    # false : flip 0.1;

  exam_grade =
```

```
    case <understands, student.good_test_taker> of
    # <true, true> : dist [ 0.6 : 'A, 0.3 : 'B, 0.1 : 'C ]
    # <true, false> : dist [ 0.4 : 'A, 0.2 : 'B, 0.4 : 'C ]
    # <false, true> : dist [ 0.5 : 'A, 0.2 : 'B, 0.3 : 'C ]
    # <false, false> : dist [ 0.1 : 'A, 0.4 : 'B, 0.5 : 'C ];

  homework_grade =
    case <understands, student.hard_working> of
    # <true, true> : dist [ 0.6 : 'A, 0.3 : 'B, 0.1 : 'C ]
    # <true, false> : dist [ 0.4 : 'A, 0.2 : 'B, 0.4 : 'C ]
    # <false, true> : dist [ 0.5 : 'A, 0.2 : 'B, 0.3 : 'C ]
    # <false, false> : dist [ 0.1 : 'A, 0.4 : 'B, 0.5 : 'C ];
};

field1 = field();

prof1 = prof(basket_weaving);
prof2 = prof(field1);

course1 = course(prof1, o_chem);
course2 = course(prof2, field1);

student1 = student()
```

■

Example 3.3: Markov chains can easily be encoded in IBAL. Here we present an example where the states are integers. The sequence of chains produced by the model is represented as a `List`. The first line of the program defines the `List` data type:

```
data List [a] = Nil | Cons (a, List [a])
```

This declaration states that `List` is a parameterized type, taking on the type parameter `a`. That is, for any type `a`, `List [a]` is also a type. It then goes on to state that a `List [a]` can be one of two things: it can be `Nil`, or it can be the `Cons` of two arguments, the first of type `a` and the second of type `List [a]`.

Given a sequence of states represented as a `List`, it is useful to be able to examine a particular state in the sequence. The standard function `nth` does this.

```
nth (n,l) : (Int, List [a]) -> a =
  case l of
  # Cons (x,xs) : if n==0 then x else nth (n-1,xs)
  # Nil : error "Too short";
```

The first line of `nth` includes a typing rule. It states that `nth` is a function taking two arguments, where the first is an integer and the second is a `List [a]`, and returning a value of type `a`.

Next, we define the types to build up a Markov model. A Markov model consists of two functions, an initialization function and a transition function. The initialization function takes zero arguments and produces a state. The transition function takes a state argument and produces a state. Markov models are parameterized by the type of the state, which is here called `a`.

```
type Init [a] = () -> a;
type Trans [a] = (a) -> a;
type Markov [a] = < init : Init [a], trans : Trans [a] >;
```

Given a Markov model, we can realize it to produce a sequence of states.

```
realize (m) : (Markov [a]) -> List [a] =
  let f(x) = Cons (x, f(m.trans (x))) in
  f(m.init ());
```

Thus far, the definitions have been abstract, applying to every Markov model. Now we define a particular Markov model by supplying definitions for the initialization and transition functions. Note that the state here is integer, so the state space is infinite. The state can be any type whatsoever, including algebraic data types like lists or trees.

```
random_walk : Markov [Int] =
     < init : fun () -> 0,
       trans : fun (n) -> dist [ 0.5 : n++, 0.5 : n-- ] >;
```

It is easy to see how to generalize this example to Hidden Markov Models by providing an observation function, and then specifying the observations using `obs` expressions. Then, combined with the previous example of Bayesian networks, we can generalize this to dynamic Bayesian networks (Dean & Kanazawa, 1989). Finally, combining with the PRM example, we can generalize to dynamic probabilistic relational models (DPRMs) (Sanghai, Domingos, & Weld, 2003). ∎

Example 3.4: We now present two examples using IBAL's decision making constructs. The first example presents an influence diagram (Howard & Matheson, 1984). The second example extends this to a Markov decision process (MDP) (Bellman, 1957). The examples involve a driver who needs to decide whether or not to speed. Speeding increases the probability of arriving on time, but runs the risk of incurring a large negative reward from being caught, which also makes it impossible to arrive on time. Besides making the decision about speeding, the decision maker also gets to decide whether to spend energy looking for a cop. The full influence diagram is as follows:

```
cop = flip 0.1;
decide look_for_cop;
see_cop =
  if look_for_cop
  then cop & flip 0.6
```

```
  else cop & flip 0.3;
decide speed given see_cop;
slow_traffic = if cop then flip 0.8 else flip 0.3;
ticket = cop & speed & flip 0.5;
on_time =
  case <ticket,speed,slow_traffic> of
  # <true, _, _> : false
  # <false, true, false> : flip 0.9
  # <false, true, true> : flip 0.5
  # <false, false, false> : flip 0.3
  # <false, false, true> : flip 0.1
reward case ticket of
  # true : -100.0
  # false : 0.0
reward case on_time of
  # true : 10.0
  # false : -10.0
reward case look_for_cop of
  # true : -1.0
  # false : 0.0
```

To turn this example into an MDP, we now make it staged decision problem, where the goal is to arrive early at the end of the last stage. Note the use of `discount` with a recursive call to the MDP function, to specify that the reward at any stage is the local reward plus the discounted future reward. The entire MDP is as follows:

```
mdp(n, state) =
  if n == 0
  then
    if state == 'early
    then reward 10.0 in 'done
    else reward -10.0 in 'done
  else
    cop = flip 0.1;
    decide look_for_cop;
    see_cop =
      if look_for_cop
      then cop & flip 0.6
      else cop & flip 0.3
    decide speed given see_cop;
    ticket = cop & speed & flip 0.5;
    new_state =
      case <ticket,speed,state> of
      # <true, _, _> : 'late
      # <false, true, 'early> : dist [ 0.9 : 'early, 0.1 : 'late ]
      # <false, true, 'late> : dist [ 0.6 : 'early, 0.4 : 'late ]
```

```
    # <false, false, 'early> : dist [ 0.7 : 'early, 0.3 : 'late ]
    # <false, false, 'late> : dist [ 0.1 : 'early, 0.9 : 'late ]
  reward case ticket of
    # true : -60.0
    # false : 0.0
  reward case look_for_cop of
    # true : -1.0
    # false : 0.0
  continuation = discount 0.95 in mdp(n-1, new_state)
```

It is easy to convert this MDP example to an infinite horizon MDP by getting rid of the first argument to the function `mdp`. However, IBAL's inference algorithm only works on finite portions of infinite models. It would not exactly simulate infinite horizon value iteration. See Section 8 for a discussion of how IBAL might be made to capture infinite horizon problems. ∎

Example 3.5: One of the features of PRMs is *structural uncertainty*: uncertainty over the relational structure of the domain. One kind of structural uncertainty is *number uncertainty*, where we do not know how many objects an object is related to by a particular relation. In the development of the SPOOK system (Pfeffer, Koller, Milch, & Takusagawa, 1999), a good deal of code was devoted to handling number uncertainty. In this example, I show how to encode number uncertainty in IBAL. By encoding it in IBAL, a lot of code is saved, and all the inference mechanisms for dealing with number uncertainty are essentially attained for free.

The main mechanism for representing number uncertainty in IBAL is a function `create` that creates a set consisting of a given number of objects of a certain kind. In addition to the number of objects, the function takes the function used to create individual objects as an argument:

```
create(n,f) =
  if n = 0
  then Nil
  else Cons(f(), create(n-1, f))
```

In this function, the argument `f` is a function that takes zero arguments. However, `create` can easily be used to create objects when the creating function takes arguments, by passing an intermediate function as follows. In the following code snippet, the `field` argument is the same for every course that is created, but the `prof` argument is different. We see here that the functional framework provides a great deal of flexibility in the way arguments are defined and passed to functions.

```
let f() =
  let p = prof(field1) in
  course(p, field)
in
create(5, f)
```

Once we have defined how to create sets of a given size, we can easily introduce uncertainty over the size. The number of objects to create is defined by its own expression, which may include `dist` or `uniform` expressions.

After creating a set, we want to be able to talk about properties of the set. PRMs use aggregate operators for this, and these can easily be encoded in IBAL. The following `count` function counts how many members of a set satisfy a given property. The first argument `p` is a predicate that takes an element of the set as argument and returns a Boolean.

```
count(p, s) =
  case s of
  # Nil : 0
  # Cons(x,xs) :
      if p x
      then 1 + count(p, xs)
      else count(p, xs)
```

In addition to `count`, we can easily define universal and existential quantifiers and other aggregates. ∎

Example 3.6: IBAL is an ideal language in which to rapidly prototype new probabilistic models. Here we illustrate using a recently developed kind of model, the *repetition model* (Pfeffer, 2004). A repetition model is used to describe a sequence of elements in which repetition of elements from earlier in the sequence is a common occurence. It is attached to an existing sequence model such as an n-gram or an HMM. Here we describe the repetition HMM.

In a repetition HMM, there is a hidden state that evolves according to a Markov process, just as in an ordinary HMM. An observation is generated at each time point. With some probability ρ, the observation is generated from memory, meaning that a previous observation is reused for the current observation. With the remaining $1 - \rho$ probability, the observation is generated from the hidden state according to the observation model of the HMM. This model captures the fact that there is an underlying generative process as described by the HMM, but this process is sometimes superseded by repeating elements that have previously appeared. Repetition is a key element of music, and repetition models have successfully been applied to modeling musical rhythm.

To describe a repetition HMM in IBAL, we first need a function to select a random element from a sequence. The function `nth` takes an integer argument and selects the given element of the sequence. We then let the argument range uniformly over the length of the sequence, which is passed as an argument to the `select` function.

```
nth(n, seq) =
  case seq of
  # Cons(x,xs) :
      if n = 0
      then x
      else nth(n-1, xs)
  # Nil : error
```

```
select(length, seq) = nth(uniform length, seq)
```

Similar to the way we defined Markov modelss earlier, a repetition HMM takes `init`, `trans` and `obs` functions as arguments. The parameter ρ is declared to be a learnable parameter. The generation process is described exceedingly simply. A function `sequence` generates the sequence of observations using a given memory of a given length, under the given model, beginning in a given state. The first thing it does is generate the new hidden state according to the transition model. Then it generates the observation using a `pdist` expression on the parameter ρ. With probability ρ it selects the observation from memory, otherwise it uses the observation model. Finally, the entire sequence is put together by consing the current observation with the new sequence formed using the new memory from the new hidden state.

```
type Init [a] = () -> a;
type Trans [a] = (a) -> a;
type Obs [a,o] = (a) -> o;
type repetition_hmm [a,o] =
  < init : Init [a], trans : trans [a], obs : Obs [a,o] >;

param rho = [ 0.1, 0.9 ];

sequence(memory, length, model, state) =
  let h = model.trans(state) in
  let o = pdist rho [ select(length, memory), model.obs(h) ] in
  Cons(o, sequence(Cons(o, memory), length + 1, model, h))

repetition_hmm(model) =
  sequence(Nil, 0, model, state.init())
```

4. Semantics

In specifying the semantics of the language, it is sufficient to provide semantics for the core expressions, since the syntactic sugar is naturally induced from them. The semantics of expressions will be provided in two ways. The first is through a sampling semantics. This semantics will formally describe a stochastic experiment that randomly generates values. The meaning of an expression is then the probability distribution over values generated. This semantics corresponds closely to the informal intuitions used in presenting the language.

This sampling semantics is the simplest and most intuitive way to understand IBAL programs. However, it only specifies the meaning of programs indirectly, by describing a stochastic experiment. Furthermore, as we shall see, it breaks down in describing the meaning of decision problems. Therefore we provide a second, more direct, semantics. In this semantics the meaning of a program is specified directly in terms of a probability

distribution over values. As with the sampling semantics, the meaning of an expression depends on the environment, but now the semantics takes a probability distribution over environments as its argument.

4.1 Sampling Semantics

The sampling semantics is presented through a sampling function §. To define the sampling semantics, we need the standard notion from programming languages of an environment. An environment is a collection of bindings mapping variable names to values. When a variable is defined during the course of a program, a binding mapping its name to its value is added to the environment. When a variable name is referred to in a program, its value is retrieved from the environment. We use the notation $\epsilon[x/v]$ to denote the operation of extending environment ϵ by binding x to v, and the notation $\epsilon(x)$ to denote the value assigned to x in ϵ.

In IBAL, two environments are used. The first, which will simply be called an environment, is a standard one, binding ordinary variable names to values. The second, a parameter environment, binds probabilistic parameters to their values. The sampling function §takes an expression, an ordinary environment, and a parameter environment, and produces a value. The notation $\S[e]\ \epsilon\ \pi$ will mean the value produced by sampling expression e in environment ϵ and parameter environment π.

We define an IBAL *program* to be an expression with no free variables or probabilistic parameters. The meaning of a program does not depend on an environment. Under the sampling semantics, the meaning of a program is the probability distribution over values produced by applying §to the program in the empty environment.

We now proceed to specify the sampling function for different kinds of expressions. Most of these rules are the same as for an ordinary functional programming language. In defining the semantics, we assume that all programs are well-typed, so that, for example, variables are always bound before they are referred to, and the function to apply in an application always has the right number of arguments. For a constant v, the sampling rule is simply

$$\S[v]\ \epsilon\ \pi = v$$

The sampling rule for `if` expressions is

$$\S[\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\ \epsilon\ \pi = \texttt{if}\ (\S[e_1]\ \epsilon\ \pi)\ \texttt{then}\ (\S[e_2]\ \epsilon\ \pi)\ \texttt{else}\ (\S[e_3]\ \epsilon\ \pi)$$

The sampling rule for `dist` expressions is

$$\S[\texttt{dist}[p_1 : e_1, \ldots, p_n : e_n]]\ \epsilon\ \pi = \S[e_i]\ \epsilon\ \pi \text{ with probability } p_i$$

The sampling rule for accessing a previously bound variable is very simple:

$$\S[x]\ \epsilon\ \pi = \epsilon(x)$$

The rule for defining a variable using a `let` expression is as follows:

$$\S[\texttt{let}\ x = e_1\ \texttt{in}\ e_2]\ \epsilon\ \pi = \S[e_2]\ \epsilon[x/\S[e_1]\ \epsilon\ \pi]\ \pi$$

where $\epsilon[x/\S[e_1]\ \epsilon\ \pi]$ denotes the extension of environment ϵ by binding x to the result of sampling e_1 in ϵ and π.

The value produced by a `lambda` or `fix` expression is a closure of the form $\{\text{args} = x_1, \ldots, x_n; \text{body} = e; \text{env} = \epsilon; \text{paramenv} = \pi\}$ specifying the arguments to the function, the body expression, the environment in which to evaluate free variables in the function, and the parameter environment in which to evaluate probabilistic parameters appearing in the function. The sampling rules for `lambda` and `fix` expressions are distinguished by the way they define the closure environment.

$$\S[\texttt{lambda}\ x_1, \ldots, x_n \rightarrow e]\ \epsilon\ \pi = \{\text{args} = x_1, \ldots, x_n; \text{body} = e; \text{env} = \epsilon; \text{paramenv} = \pi\}$$

$$\S[\texttt{fix}\ f(x_1, \ldots, x_n) \rightarrow e]\ \epsilon\ \pi = v$$
$$\text{where } v = \{\text{args} = x_1, \ldots, x_n; \text{body} = e; \text{env} = \epsilon[f/v]; \text{paramenv} = \pi\}$$

In the sampling rule for function application, we sample the expression defining the function to be applied, and the expressions for all the arguments, independently. We then sample the body of the function, in an environment created by extending the closure environment with the values of the arguments.

$$\S[e_0(e_1, \ldots, e_n)]\ \epsilon\ \pi =$$
$$\S[e]\ \epsilon'[x_1/\S[e_1]\ \epsilon, \ldots, x_n/\S[e_n]\ \epsilon]\ \pi'$$
$$\text{where } \{\text{args} = x_1, \ldots, x_n; \text{body} = e; \text{env} = \epsilon'; \text{paramenv} = \pi'\} = \S[e_0]\ \epsilon\ \pi$$

The sampling rule for tuple construction is

$$\S[< x_1 : e_1, \ldots, x_n : e_n >]\ \epsilon\ \pi =< x_1 : \S[e_1]\ \epsilon\ \pi, \ldots, x_n : \S[e_n]\ \epsilon\ \pi >$$

For extracting a component from a tuple, the sampling rule is simply

$$\S[e.x]\ \epsilon\ \pi = \text{the component labeled } x \text{ in } (\S[e]\ \epsilon\ \pi)$$

Recall that an observation serves to condition the probability distribution defined by a model on the observation being true. This is accomplished through rejection sampling semantics. The meaning of

$$\texttt{obs}\ x = v\ \texttt{in}\ e$$

in terms of a stochastic experiment, is to begin by testing whether x has previously been assigned the value v. If it has, the expression e is evaluated, otherwise the whole experiment is rejected. The meaning of an expression under the sampling semantics is now not simply the probability distribution over values generated by sampling the expression, but the distribution over values conditioned on the experiment not being rejected. This has the desired effect that the resulting distribution is the distribution over values conditioned on the observation being true.

The sampling function from the previous section is extended as follows:

$$\S[\texttt{obs}\ x = v\ \texttt{in}\ e]\ \epsilon\ \pi = \begin{cases} \bot & \text{if } \epsilon(x) \neq v \\ \S[e]\ \epsilon\ \pi & \text{if } \epsilon(x) = v \end{cases}$$

The symbol $\bot$ denotes a sample that is rejected. The meaning of a program is now the probability distribution over values produced by $\S$, conditioned on the value not being $\bot$.

In the sampling rules presented thus far, the parameter environment π played no interesting role. Its purpose is to allow the rules for learnable parameters to be formulated. The rule for a `pdist` expression is as follows:

$$\S[\texttt{pdist}\ x\ [e_1, \ldots, e_n]]\ \epsilon\ \pi = \S[e_i]\ \epsilon\ \pi \text{ with probability } \pi(x_i)$$

The rule for extending an environment on encountering a `param` expression is

$$\S[\texttt{param}\ x = \alpha_1, \ldots, \alpha_n\ \texttt{in}\ e]\ \epsilon\ \pi = \S[e]\ \epsilon\ \pi[x_1/\theta_1, \ldots, x_n/\theta_n)]$$
$$\text{where } \theta_1, \ldots, \theta_n \sim \text{Dirichlet}(\alpha_1, \ldots, \alpha_n)$$

In a `choose` expression, the value of the decision variable is the one that maximizes the expected utility of the result expression. In order to capture this, we need to define an expected utility of an expression. The sampling semantics breaks down here, because it only deals with one particular run of the experiment. To talk about the expected utility, we need to consider all runs of the experiment.

4.2 Distributional Semantics

We use the notation $\mathcal{M}[e]$ to denote the meaning of expression e, under the distributional semantics. The meaning function takes as argument a joint probability density function over environments and parameter environments. The function returns a probability distribution over values. We write $\mathcal{M}[e]\ \Delta\ v$ to denote the probability of v under the meaning of e when the probability density function over environments and parameter environments is Δ. We also use the notation $\mathcal{M}[e]\ (\epsilon, \pi)\ v$ to denote the probability of v under the meaning of e when the density function over environments assigns positive probability only to ϵ and π. The reason why the meaning function takes a probability distribution over environments rather than a single environment will become clear later, when we describe the semantics of decision problems in IBAL.

We now define the meaning function for different types of expressions. The meaning of a constant expression is given by

$$\mathcal{M}[v]\ \Delta\ v' = \begin{cases} 1 & \text{if v' = v} \\ 0 & \text{otherwise} \end{cases}$$

The meaning of an `if` expression is defined as follows. We first take the sum over all environments of the meaning of the expression in the particular environments. The reason we need to do this is because the meanings of the `if` clause and of the `then` and `else` clauses are correlated by the environment. Therefore we need to specify the particular environment before we can break up the meaning into the meanings of the subexpressions. Given the environments, however, the subexpressions become conditionally independent, so we can multiply their meanings together.

$$\mathcal{M}[\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\ \Delta\ v = \int_{\epsilon,\pi} \Delta(\epsilon, \pi) \begin{bmatrix} (\mathcal{M}[e_1]\ (\epsilon, \pi)\ \texttt{true})(\mathcal{M}[e_2]\ (\epsilon, \pi)\ v) + \\ (\mathcal{M}[e_1]\ (\epsilon, \pi)\ \texttt{false})(\mathcal{M}[e_3]\ (\epsilon, \pi)\ v) \end{bmatrix}$$

The distributional semantics of a `dist` expression simply states that the probability of a value under a `dist` expression is the weighted sum of the probability of the value under

the different branches:

$$\mathcal{M}[\texttt{dist}[p_1 : e_1, \ldots, p_n : e_n]]\ \Delta\ v = \sum_i p_i(\mathcal{M}[e_i]\ \Delta\ v)$$

The probability that referring to a variable produces a value is obtained simply by summing over environments in which the variable has the given value:

$$\mathcal{M}[x]\ \Delta\ v = \int_{\epsilon,\pi:\epsilon(x)=v} \Delta(\epsilon, \pi)$$

To define the meaning of a `let` expression, we consider extending all possible environments ϵ by all possible values v'. We multiply the probability of getting v' under e_1 in ϵ with the probability of getting v under e_2 in the extended environment:

$$\mathcal{M}[\texttt{let}\ x = e_1\ \texttt{in}\ e_2]\ \Delta\ v = \int_{\epsilon,\pi} \Delta(\epsilon, \pi) \sum_{v'} (\mathcal{M}[e_1]\ (\epsilon, \pi)\ v')(\mathcal{M}[e_2]\ (\epsilon[x/v'], \pi)\ v)$$

The meaning of a `lambda` expression is given by

$$\mathcal{M}[\texttt{lambda}\ x_1, \ldots, x_n \to e]\ \Delta\ v = \begin{cases} \Delta(\epsilon, \pi) & \text{if } v = \left\{ \begin{array}{l} \text{args} = x_1, \ldots, x_n; \\ \text{body} = e; \\ \text{env} = \epsilon; \\ \text{paramenv} = \pi \end{array} \right\} \\ 0 & \text{otherwise} \end{cases}$$

The meaning of `fix` is very similar, except that the function name is bound in the environment.

$$\mathcal{M}[\texttt{fix}\ x_1, \ldots, x_n \to e]\ \Delta\ v = \begin{cases} \Delta(\epsilon, \pi) & \text{if } v = \left\{ \begin{array}{l} \text{args} = x_1, \ldots, x_n; \\ \text{body} = e; \\ \text{env} = \epsilon[f/v]; \\ \text{paramenv} = \pi \end{array} \right\} \\ 0 & \text{otherwise} \end{cases}$$

The distributional semantics for function application is logically constructed as follows. We sum over all possible environments, and over all possible values $v_0, v_1, \ldots, v_n$, of the expression e_0 defining the function to be applied, and of the expressions $v_1, \ldots, v_n$ defining the arguments. We take the product of the probabilities of obtaining each v_i from e_i in the environment, and multiply by the probability that applying v_0 to $v_1, \ldots, v_n$ produces the value v. Here, applying v_0 to $v_1, \ldots, v_n$ means taking the meaning of the body of v_0 in an environment formed by extending the closure environment by binding argument x_i to v_i.

$$\mathcal{M}[e_0(e_1, \ldots, e_n)]\ \Delta\ v =$$
$$\int_{\epsilon,\pi} \Delta(\epsilon, \pi) \sum_{v_0, v_1, \ldots, v_n} (\prod_{i=0}^{n} \mathcal{M}[e_i]\ \epsilon\ v_i)(\mathcal{M}[e]\ (\epsilon'[x_1/v_1, \ldots, x_n/v_n], \pi')\ v)$$
$$\text{where } \{\text{args} = x_1, \ldots, x_n; \text{body} = e; \text{env} = \epsilon'; \text{paramenv} = \pi'\} = v_0$$

The meaning of a tuple expression is given by

$$\mathcal{M}[< x_1 : e_1, \ldots, x_n : e_n >]\ \Delta\ v =$$
$$\begin{cases} \int_{\epsilon,\pi} \Delta(\epsilon, \pi) \prod_{i=1}^{n} \mathcal{M}[e_i]\ (\epsilon, \pi)\ v_i & \text{if } v =< x_1 : v_1, \ldots, x_n : v_n > \\ 0 & \text{otherwise} \end{cases}$$

The meaning of extracting a component from a tuple is

$$\mathcal{M}[e.x]\ \Delta\ v = \sum_{v':v'.x=v} \mathcal{M}[e]\ \Delta\ v'$$

The distributional semantics captures observations quite simply and elegantly. The effect of an observation is to condition the distribution Δ over environments on the observation holding.

$$\mathcal{M}[\texttt{obs}\ x = v\ \texttt{in}\ e]\ \Delta\ v = \frac{\int_{\epsilon,\pi:\epsilon(x)=v} \Delta(\epsilon,\pi)(\mathcal{M}[e]\ (\epsilon,\pi)\ v)}{\int_{\epsilon,\pi:\epsilon(x)=v} \Delta(\epsilon,\pi)}$$

The meaning of expressions defining new probabilistic parameters is obtained by considering extending all possible parameter environments with all possible parameter values.

$$\mathcal{M}[\texttt{param}\ x = \alpha_1,\ldots,\alpha_n\ \texttt{in}\ e]\ \Delta\ v =$$
$$\int_{\epsilon,\pi} \Delta(\epsilon,\pi) \int_{\theta_1,\ldots,\theta_n} f(\theta_1,\ldots,\theta_n)(\mathcal{M}[e]\ (\epsilon,\pi[x_1/\theta 1,\ldots,x_n/\theta_n])\ v)$$
where $f(\theta_1,\ldots,\theta_n)$ is defined by Dirichlet$(\alpha_1,\ldots,\alpha_n)$

To get the meaning of `pdist`, we need to look at the value of the parameter in the parameter environment, and select each branch with the appropriate probability.

$$\mathcal{M}[\texttt{pdist}\ x\ [p_1 : e_1,\ldots,p_n : e_n]]\ \Delta\ v = \int_{\epsilon,\pi} \Delta(\epsilon,\pi) \sum_{i=1}^{n} \pi(x_i)(\mathcal{M}[e_i]\ (\epsilon,\pi)\ v)$$

Once we have `reward` expressions, the meaning of an expression is not just the probability distribution over values it produces, but also an expected utility. We now define the expected utility of all expression types. The expected utility function $\mathcal{U}$ takes a probability density function over environments and parameter environments. The basic idea behind these definitions is that in any experiment, the utility of an expression is the sum of the expected utilities of the subexpressions that are evaluated. So, for example, for an `if-then-else` expression, the utility is the utility of the `if` clause, plus the utility of either the `then` or the `else` clause, depending on the value of the `if` clause. Therefore the expected utility of the expression is the expected utility of the `if` clause, plus the expected utility of the `then` clause times the probability that the `then` clause is taken, plus the expected utility of the `else` clause times the probability that the `else` clause is taken. Most of the definitions are fairly straightforward given the previous definitions of the meaning function, so they will be presented without comment.

$$\mathcal{U}[v]\ \Delta = 0$$

$$\mathcal{U}[\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\ \Delta =$$
$$\mathcal{U}[e_1]\ \Delta + (\mathcal{M}[e_1]\ \Delta\ \texttt{true})(\mathcal{U}[e_2]\ \Delta) + (\mathcal{M}[(]e_1)\ \Delta\ \texttt{false})(\mathcal{U}[e_3]\ \Delta)$$

$$\mathcal{U}[\texttt{dist}[p_1 : e_1,\ldots,p_n : e_n]]\ \Delta = \sum_{i=1}^{n} p_i(\mathcal{U}[e_i]\ \Delta)$$

$$\mathcal{U}[x]\ \Delta = 0$$

$$\mathcal{U}[\texttt{let}\ x = e_1\ \texttt{in}\ e_2]\ \Delta = \int_{\epsilon,\pi} \Delta(\epsilon,\pi)[\sum_{v} \mathcal{U}[e_1]\ (\epsilon,\pi) + (\mathcal{M}[e_1]\ \epsilon,\pi\ v)(\mathcal{U}[e_2]\ (\epsilon[x/v],\pi))]$$

$$\mathcal{U}[\texttt{lambda}\ x_1, \ldots, x_n \rightarrow e]\ \Delta = 0$$

$$\mathcal{U}[\texttt{fix}\ f(x_1, \ldots, x_n) \rightarrow e]\ \Delta = 0$$

$$\mathcal{U}[e_0(e_1, \ldots, e_n)]\ \Delta =$$
$$\sum_{i=0}^{n} \mathcal{U}[e_i]\ \Delta + \int_{\epsilon,\pi} \Delta(\epsilon, \pi) \sum_{v_0, v_1, \ldots, v_n} (\mathcal{M}[e_0]\ (\epsilon, \pi)\ v_0)(\mathcal{U}[e]\ (\epsilon'[x_1/v_1, \ldots, x_n/v_n], \pi'))$$
where $\{\text{args} = x_1, \ldots, x_n; \text{body} = e; \text{env} = \epsilon'; \text{paramenv} = \pi'\} = v_0$

$$\mathcal{U}[< x_1 : e_1, \ldots, x_n : e_n >]\ \Delta = \sum_{i=1}^{n} \mathcal{U}[e_i]\ \Delta$$

$$\mathcal{U}[e.x]\ \Delta = \mathcal{U}[e]\ \Delta$$

$$\mathcal{U}[\texttt{obs}\ x = v\ \texttt{in}\ e]\ \Delta = \mathcal{U}[e]\ (\Delta \mid [x = v])$$
where the notation $\Delta \mid [x = v]$ means Δ conditioned on $x = v$

$$\mathcal{U}[\texttt{param}\ x = \alpha_1, \ldots, \alpha_n\ \texttt{in}\ e]\ \Delta =$$
$$\int_{\epsilon,\pi} \Delta(\epsilon, \pi) \int_{\theta_1, \ldots, \theta_n} f(\theta_1, \ldots, \theta_n)(\mathcal{U}[e]\ (\epsilon, \pi[x_1/\theta 1, \ldots, x_n/\theta_n]))$$
where $f(\theta_1, \ldots, \theta_n)$ is defined by Dirichlet$(\alpha_1, \ldots, \alpha_n)$

$$\mathcal{U}[\texttt{pdist}\ x\ [p_1 : e_1, \ldots, p_n : e_n]]\ \Delta = \sum_{i=1}^{n} \int_{\epsilon,\pi} \Delta(\epsilon, \pi)\pi(x_i)(\mathcal{U}[e_i]\ (\epsilon, \pi))$$

$$\mathcal{U}[\texttt{reward}\ r\ \texttt{in}\ e]\ \Delta = r + \mathcal{U}[e]\ \Delta$$

Now that we have defined the expected utility of an expression, we are ready to define the meaning of `choose` expressions. In defining the semantics, we need to handle the information available to the decision maker correctly. The first point is that we need to enforce the "no forgetting" rule of influence diagrams. This rule states that any previous decisions, as well as all information that was known at the time a previous decision was made, is available at the time a later decision is made. In other words, one cannot forget information one had at earlier decisions.

In IBAL, we make the added assumption that within any function, the arguments of the function are always known to the decision maker. This is to ensure that a function can be treated as a black box, representing a conditional distribution from the function arguments to output values. If we did not assume that the arguments to the function were always known, the function's meaning would vary depending on which arguments are known for a specific function call. Therefore the function would not be a black box; it would be dependent on the code of the calling function.

The no forgetting rule, in addition to the rule that function arguments are available information, can be handled simply by syntactically adding all the extra available information to the list of given information. In the definition below of the meaning of a `choose` expression, we assume that this syntactic modification has been made.

Now, in evaluating the expected utility of the result expression, the variables in the environment that are part of the information are given, but other variables in the environment are not known. Therefore the expected utility is computed according to a distribution over the variables in the environment not in the information, conditioned on the variables in the information. For this reason, we need a distribution over environments. This is why the meaning function takes a distribution over environments as its argument, rather than a single environment. The meaning is defined as follows:

$$\mathcal{M}[\texttt{choose } x \texttt{ from } s_1, \ldots, s_n \texttt{ given } x_1, \ldots, x_m \texttt{ in } e]\ \Delta\ v = \int_{\epsilon,\pi} \Delta(\epsilon, \pi)(\mathcal{M}[e]\ (\epsilon[x/v(\epsilon)], \pi)\ v)$$
$$\text{where } v(\epsilon) = \arg\max_{s \in s_1,\ldots,s_n} \mathcal{U}[e](\Delta \mid [x_1 = \epsilon(x_1), \ldots, x_n = \epsilon(x_n)])[x/s]$$

Here, the notation $(\Delta \mid [x_1 = \epsilon(x_1), \ldots, x_n = \epsilon(x_n)])$ means the distribution over environments Δ conditioned on the variables $x_1, \ldots, x_n$ taking on their respective values $\epsilon(x_1), \ldots, \epsilon(x_n)$ in the environment. The notation $(\Delta \mid [x_1 = \epsilon(x_1), \ldots, x_n = \epsilon(x_n)])[x/s]$ means the distribution P over environments in which $P(\epsilon')$ is equal to $(\Delta \mid [x_1 = \epsilon(x_1), \ldots, x_n = \epsilon(x_n)])(\epsilon)$ when ϵ' is equal to ϵ extended by binding x to s. In other words, all environments that receive positive probability under $(\Delta \mid [x_1 = \epsilon(x_1), \ldots, x_n = \epsilon(x_n)])$ are extended by binding x to s. It is this distribution over extended environments that is used to evaluate the expected utility of the result expression. We also define the utility of `choose` expressions analogously.

$$\mathcal{U}[\texttt{choose } x \texttt{ from } s_1, \ldots, s_n \texttt{ given } x_1, \ldots, x_m \texttt{ in } e]\ \Delta = \int_{\epsilon,\pi} \Delta(\epsilon, \pi)(\mathcal{U}[e]\ (\epsilon[x/v(\epsilon)], \pi))$$

4.3 Lazy Semantics

A very natural way to define a probabilistic model is to describe a generative model that generates possibly infinite values, and then to ask queries that only consider a finite portion of the values. For example, a stochastic context free grammar may generate arbitrarily long strings. We may query the probability that a grammar generates a particular string. This requires looking at only a finite portion of the generated string, and a finite portion of the generation process.

We use lazy semantics in IBAL to get at the idea that only those parts of an expression that need to be evaluated in order to generate the result of the expression are evaluated. There are two places in particular where this applies. In a

$$\texttt{let } x_1 = e_1 \texttt{ in } e_2$$

expression, the subexpression e_1 is only evaluated if x is actually needed in evaluating e_2. More precisely, if e_1 defines a tuple, only those components of the tuple that are needed in e_2 are evaluated. In a function application

$$e_0(e_1, \ldots, e_n)$$

only those parts of the argument e_i are evaluated that are needed in evaluating the body of the function. The body of the function here could mean the body of any possible value of e_0 — if any value of e_0 requires a component of the argument, the component is evaluated.

Example 4.1: Consider the program

```
f() = Cons(flip 0.5, f())

g(x) =
  case x of
  # Cons(y,z) -> y

g(f())
```

The function `f()` defines an infinite sequence of `true` and `false` elements. The function `g()` then returns the first element in the sequence. When `g` is applied to `f`, the body of `g` specifies that only the first component of its argument is required. Therefore, when evaluating `f`, only its first component will be evaluated. That can be done by examining a single `flip`. ∎

5. Desiderata for Inference

IBAL is able to capture many traditional kinds of representations, such as Bayesian networks (BNs), hidden Markov models (HMMs) and stochastic context-free grammars (SCFGs). It can also express more recent models such as object-oriented Bayesian networks (OOBNs) and relational probability models (Pfeffer, 2000). For IBAL to be successful as a general-purpose language, the implementation should be designed to capture effective strategies for as many models as possible. This leads to the following desiderata. To be sure, this list is not complete. In particular, it does not consider issues to do with the time-space tradeoff. Nevertheless, it is a good set of goals, and no existing implementation is able to achieve all of them.

1: Exploit Independence Independence and conditional independence are traditionally exploited by BNs. The inference algorithm should have similar properties to traditional BN algorithms when run on BN models.

2: Exploit Low-Level Structure In Bayesian networks, the conditional probability distribution over a variable given its parents is traditionally represented as a table. Researchers have studied more compact representations, such as noisy-or and context specific independence. Special purpose inference algorithms have been designed for these structures (Heckerman & Breese, 1994; Poole & Zhang, 2003). Because of IBAL's programming language constructs, it is easy to describe such structures — easier, in fact, than describing full conditional probability tables. The inference algorithm should be able to take a representation that elucidates the low-level structure and automatically provide benefits from exploiting the structure.

3: Exploit High-Level Structure Larger models can often be decomposed into weakly interacting components. It was discovered for OOBNs (Pfeffer et al., 1999) that exploiting such high-level structure is a big win. In particular, the different components tend to be largely decoupled from one another, and they can be separated by a small interface that renders them conditionally independent of each other. IBAL represents high-level structure using functions. The inference algorithm should take advantage of the decoupling of the internals of functions from the remainder of the model.

4: Exploit Repetition Many frameworks such as SCFGs and HMMs involve many repeated computations. In IBAL, the same function can be applied many times, and this should be exploited to avoid repeated computation.

5: Exploit the Query Often, one describes a very complex or infinite probabilistic model, but asks a query that only requires a small portion of the model. This is the process, for example, for SCFGs: the grammar can generate arbitrarily long sentences, but only a finite generation process is needed to generate a particular finite sentence. IBAL should use the query to consider only the parts of the generation process that are necessary for producing the result.

6: Exploit Support of Variables When a model contains variables, its behavior depends on their values. The *support* of a variable is the set of values it can take with positive probability. Taking the support into account can simplify inference, by restricting the set of inputs that need to be considered. It can turn a potentially infinite inference into a finite one.

7: Exploit Evidence If we have observations in the program, they can be used to further limit the set of values variables can take, and to restrict the possible computations. For example, suppose we have a model in which a string is generated by a grammar that can generate arbitrarily long strings, and then observe that the string has length at most four. We can use this observation to restrict the portion of the grammar that needs to be examined.

6. Previous Approaches

Previous approaches to inference in high-level probabilistic languages have generally fallen in four categories. On one side are approaches that use approximate inference, particularly Markov chain Monte Carlo methods (Spiegelhalter, Thomas, Best, & Gilks, 1995; Pasula & Russell, 2001). While exact inference may be intractable for many models, and approximate strategies are therefore needed, the goal of this paper is to push exact inference as far as possible.

The first generation of high-level probabilistic languages generally used the *knowledge-based model construction (KBMC)* approach (e.g. (Poole, 1993; Ngo & Haddawy, 1996; Laskey & Mahoney, 1997; Kersting & de Raedt, 2000)). In this approach, a knowledge base describes the general probabilistic mechanisms. These are combined with ground facts to produce a Bayesian network for a specific situation. A standard BN inference algorithm is then used to answer queries.

This approach generally satisfies only the first of the above desiderata. Since a BN is constructed, any independence will be represented in that network, and can be exploited by the BN algorithm. The second desideratum can also be satisfied, if a BN algorithm that exploits low-level structure is used, and the BN construction process is able to produce that structure. Since the construction process creates one large BN, any structure resulting from weakly interacting components is lost, so the third desideratum is not satisfied. Similarly, when there is repetition in the domain the large BN contains many replicated components, and the fourth desideratum is not satisfied. Satisfaction of the remaining desiderata depends on the details of the BN construction process. The most common approach is to grow the

network using backward chaining, starting at the query and the evidence. If any of these lead to an infinite regress, the process will fail.

More recent approaches take one of two tacks. The first is to design a probabilistic representation language as a programming language, whether a functional language (Koller, McAllester, & Pfeffer, 1997; Pless & Luger, 2001) or logic programming (Muggleton, 2001). The inference algorithms presented for these languages are similar to evaluation algorithms for ordinary programming languages, using recursive descent on the structure of programs. The programming language approach has a number of appealing properties. First, the evaluation strategy is natural and familiar. Second, a programming language provides the fine-grained representational Control with which to describe low-level structure. Third, simple solutions are suggested for many of the desiderata. For example, high-level structure can be represented in the structure of a program, with different functions representing different components. As for exploiting repetition, this can be achieved by the standard technique of *memoization*. When a function is applied to a given set of arguments, the result is cached, and retrieved whenever the same function is applied to the same arguments. Meanwhile, *lazy evaluation* can be used to exploit the query to make a computation simpler.

However, approaches based on programming languages have a major drawback. They do not do a good job of exploiting independence. (Koller et al., 1997) made an effort to exploit independence by maintaining a list of variables shared by different parts of the computation. The resulting algorithm is much more difficult to understand, and the solution is only partial. Given a BN encoded in their language, the algorithm can be viewed as performing VE using a particular elimination order: namely, from the last variable in the program upwards. It is well known that the cost of VE is highly dependent on the elimination order. so the algorithm is exponentially more expensive for some families of models than an algorithm that can use any order.

In addition, while these approaches suggest solutions to many of the desiderata, actually integrating them into a single implementation is difficult. For example, (Koller et al., 1997) suggested using both memoization and lazy evaluation, believing that since both were standard techniques their combination would be simple. In fact it turns out that implementing both simultaneously is considered extremely difficult![1] The final three desiderata are all variations on the idea that knowledge can be used to simplify computation. The general approach was captured by the term *evidence-finite computation* in (Koller et al., 1997). However, this catch-all term fails to capture the distinctions between the different way knowledge can be exploited. A careful implementation of the algorithm in (Koller et al., 1997) showed that it achieved termination only in a relatively small number of possible cases. In particular it failed to exploit support and observations.

The final approach to high-level probabilistic inference is to use a structured inference algorithm. In this approach, used in object-oriented Bayesian networks and relational probabilistic models (Pfeffer et al., 1999; Pfeffer, 2000), a Bayesian network fragment is provided for each model component, and the components are related to each other in various ways. Rather than constructing a single BN to represent an entire domain, inference works directly on the structured model, using a standard BN algorithm to work within each component. The approach was designed explicitly to exploit high-level structure and repetition. In

1. Simon Peyton-Jones, personal communication.

addition, because a standard BN algorithm is used, this approach exploits independence. However, it does not address the final three desiderata. An anytime approximation algorithm (Koller & Pfeffer, 2000) was provided for dealing with infinitely recursive models, but that does not satisfy the requirements of this paper.

In addition, this approach does not do as well as one might hope at exploiting low-level structure. One might rely on the underlying BN inference algorithm to exploit whatever structure it can. For example, if it is desired to exploit noisy-or structure, the representation should explicitly encode such structure, and the BN algorithm should take advantage of it. The problem with this approach is that it requires a special purpose solution for each possible structure, and high-level languages make it easy to specify new structures. A case in point is the structure arising from quantification over a set of objects. In the SPOOK system (Pfeffer et al., 1999), an object A can be related to a set of objects B, and the properties of A can depend on an aggregate property of B. If implemented naively, A will depend on each of the objects in B, so its conditional probability table will be exponential in the size of B. As shown in (Pfeffer et al., 1999), the relationship between A and B can be decomposed in such a way that the representation and inference are linear in the size of B. Special purpose code had to be written in SPOOK to capture this structure, but it is easy to specify in IBAL as described in Example 3.5, so it would be highly beneficial if IBAL's inference algorithm can exploit it automatically.

7. Inference

7.1 Inference Overview

If we examine the desiderata of Section **??**, we see that they fall into two categories. Exploiting repetition, queries, support and evidence all require avoiding unnecessary computation, while exploiting structure and independence require performing the necessary computation as efficiently as possible. One of the main insights gained during the development of IBAL's inference algorithm is that simultaneously trying to satisfy all the desiderata can lead to quite complex code. The inference process can be greatly simplified by recognizing the two different kinds of desiderata, and dividing the inference process into two phases. The first phase is responsible for determining exactly what computations need to be performed, while the second phase is responsible for performing them efficiently.

This division of labor is reminiscent of the SPI algorithm for Bayesian network inference (Li & D'Ambrosio, 1994), in which the first phase finds a factoring of the probability expression, and the second phase solves the expression using the factoring. However, there is a marked difference between the two approaches. In SPI, the goal of the first phase is to find the order in which terms should be multiplied. In IBAL, the first phase determines which computations need to be performed, but not their order. That is left for the variable elimination algorithm in the second phase. Indeed, SPI could be used in the second phase of IBAL as the algorithm that computes probabilities.

The first phase of IBAL operates directly on programs, and produces a data structure called the *computation graph*. This rooted directed acyclic graph contains a node for every distinct computation to be performed. A computation consists of an expression to be evaluated, and the supports of free variables in the expression. The computation graph

contains an edge from one node to another if the second node represents a computation for a subexpression that is required for the first node.

The second phase of the algorithm traverses the computation graph, *solving* every node. A solution for a node is a conditional probability distribution over the value of the expression given the values of the free variables, assuming that the free variables have values in the given supports. The solution is computed bottom-up. To solve a node, the solutions of its children are combined to form the solution for the node.

On the surface, the design seems similar to that of the KBMC approaches. They both create a data structure, and then proceed to solve it. The IBAL approach shares with KBMC the idea of piggy-backing on top of existing BN technology. However, the two approaches are fundamentally different. In KBMC, the constructed BN contains a node for every random variable occuring in the solution. By contrast, IBAL's computation graph contains a node for every *distinct* computation that is performed during the solution process. If different random variables share the same computation, only one node is created. Secondly, the computation graph is not a BN. Rather, it is an intermediate data structure that guides the construction of many different BNs, and their combination to provide the solution to the query. Thirdly, traditional KBMC approaches typically do not utilize all the information available in the query, support and evidence in constructing the BN, whereas IBAL uses all of these in constructing the computation graph.

7.2 First Phase

7.2.1 Observation Propagation

It is the task of the first phase to construct the computation graph, containing a node for every computation that has to be performed. At the end of the phase, each node will contain an expression to be evaluated, annotated with the supports of the free variables, and the support of the expression itself. The first phase begins by propagating observations to all subexpressions that they effect. The result of this operation is an annotated expression, where each expression is annotated with the effective observation about its result. When the computation graph is later constructed, the annotations will be used to restrict the supports of variables, and possibly to restrict the set of computations that are required. Thus the **seventh desideratum of exploiting evidence** is achieved.

In general, an observation can be propagated in two directions. Consider

$$\texttt{let x =}\ e_1\ \texttt{in observe x = p in}\ e_2$$

Since the observation is about `x`, and `x` is defined by e_1, we can infer *backwards* that the outcome of e_1 must satisfy the observation. Suppose that e_1 defines a stochastic grammar that produces arbitrarily long sentences, and the observation is that the length of `x` is 4. Using the observation, we only need to examine realizations of e_1 that have length 4. The observation on `x` can also be propagated *forwards* to e_2, if e_2 mentions `x`. For example, suppose e_2 is a predicate testing whether `x` is longer than 10. We can immediately infer the observation that e_2 is false. This observation may itself be propagated elsewhere. Similarly, the observation that e_1 has a result of length 4 might produce additional observations about free variables in e_1.

IBAL's observation propagation takes an expression, and previously known constraints on free variables that may appear in the expression, and constraints on the result of the

expression. It returns constraints on free variables appearing in the expression and on its result derived from examining the expression itself. For example, for an expression of the form `let x =` e_1 `in` e_2, with previous constraints c on free variables, and previous constraint d on the result, it works as follows:

1. Call observation propagation on e_2, with previous constraints c and d. Let c' and d' be the resulting constraints on free variables and result. Let c'_x and c'_{-x} be the components of c' on x and other variables, respectively. (Each may be empty.)
2. Call observation propagation on e_1, with previous constraints c on free variables, and c'_x on the outcome. Let c'' and d'' be the result.
3. If d'' differs from c'_x, it means that additional constraints on the value of `x` were derived by examining e_1. We examine e_2 once more using these additional constraints.
4. The final returned constraints for the entire let expression are c'_{-x} and d' resulting from the last examination of e_2. The reason c'_{-x} is returned and not c' is that `x` is not a free variable in the entire `let` expression.

Let us see what happens in our example SCFG program from Example 2.7. Suppose we observe that the output of `s()` is at most 2. Nothing will be done with this observation until the body of `s` is expanded. At that point, the observation will be propagated to the output of `append`. When `append` is expanded, the observation is propagated to the `case` expression which is really a sequence of nested `if`s. Eventually it will be propagated to each of the consequents. In the first case, we get the constraints that `x` has length zero and `y` has length at most 2. In the second case we get the constraint that `append(z,y)` has length at most one. This constraint is in turn propagated when `append` is expanded, and the process recurses. Eventually, the recursion terminates, when we know that the result of `append` has length zero, because in that case we can infer that only the first case can hold, and both arguments have length zero. As the process filters up, we eventually infer that when the output of `append` has length at most 2, both arguments have length at most 2.

There is a subtlety to the observation propagation process. Evidence about the outcome of a function can be propagated into the function itself. Within the function, evidence may eventually be observed about the outcome of another function, and this process can recurse. Thus the process of propagating evidence can itself lead to an infinite regress, even for a computation that would terminate for other reasons. For this reason, the evidence propagation process is actually interleaved with the process of constructing the computation graph. At first, evidence is not propagated into functions, to avoid expanding functions unnecessarily. When the algorithm finally creates a node in the computation graph corresponding to a function, we know we need to expand the function, and evidence propagation is performed on the function body.

7.2.2 Lazy Memoization

After propagating observations, the process of constructing the computation graph begins. In order not to construct any more than is necessary to answer the query, the graph is constructed lazily. In particular, whenever a `let x =` e_1 `in` e_2 expression is encountered, the graph for e_2 is constructed to determine how much of `x` is required for e_2. Then only the required amount of the graph for e_1 is constructed. (Recall that a variable can have a

complex value, so only part of its value may be required in another expression.) Similarly, when a function is applied to arguments, the graph for the arguments is constructed lazily. Since no node of the computation graph is constructed unless it has been determined that it is required for solving the query, the **fifth desideratum of exploiting the query** is achieved.

There is a subtle issue here. A node in the graph is defined by the expression and supports of the free variables. But, under lazy evaluation, the node for an expression is constructed before that for its free variables. To get around this problem, instead of explicitly storing the support of `x` in the node for expression e_2, we simply store a pointer to the node for e_1. If, during the course of expanding e_2, it is discovered that some part of the value of `x` is needed, the node for e_1 is expanded to the degree necessary. The support of the relevant part of the value of `x` is then available to the expansion of e_2.

The **fourth desideratum of exploiting repetition** is achieved by avoiding repeated nodes in the graph. In particular, when a function is applied to arguments, the same node is used as long as (1) the supports of the required parts of the arguments are the same; (2) the required components of the output are the same; and (3) the observed evidence on the output is the same. This is quite a strong property. It requires that the same node be used when the supports of the arguments are the same, even if the arguments are defined by different expressions. It also stipulates that the supports only need to be the same on the required parts of the arguments.

Unfortunately, the standard technique of memoization does not interact well with lazy evaluation. The problem is that in memoization, when we want to create a new node in the computation graph, we have to check if there is an existing node for the same expression that has the same supports for the required parts of the arguments. But we don't know yet what the required parts of the arguments are, or what their supports are. Worse yet, with lazy evaluation, we may not yet know these things for expressions that already have nodes. This issue is the crux of the difficulty with combining lazy evaluation and memoization. In fact, no functional programming language appears to implement both, despite the obvious appeal of these features.

A new evaluation strategy was developed for IBAL to achieve both laziness and memoization together. The key idea is that when the graph is constructed for a function application, the algorithm speculatively assumes that an argument is not required. If it turns out that part of it is required, enough of the computation graph is created for the required part, and the graph for the application is reconstructed, again speculatively assuming that enough of the argument has been constructed. This process continues until the speculation turns out to be correct. At each point, we can check to see if there is a previously created node for the same expression that uses as much as we think is required of the argument. At no point will we create a node or examine part of the argument that is not required.

An important detail is that whenever it is discovered that an argument to the function is required, this fact is stored in the cache. This way, the speculative evaluation is avoided if it has already been performed for the same partial arguments. In general, the cache consists of a mapping from partial argument supports to either a node in the computation graph or to a note specifying that another argument is required.

For example, suppose we have a function

```
f(x,y,z) = if x then y else z
```

where the support of `x` is `{true}`, the support of `y` is `{5,6}`, and `z` is defined by a divergent function. We first try to evaluate `f` with no arguments evaluated. We immediately discover that `x` is needed, and store this fact in the cache. We obtain the support of `x`, and attempt to evaluate f again. Now, since `x` must be true, we discover that `y` is needed, and store this in the cache. We now attempt again to evaluate `f` with the supports of `x` and `y`, and since `z` is not needed, we return with a computation node, storing the fact that when `x` and `y` have the given supports, the result is the given node. The contents of the cache after the evaluation has completed are

`f(x,y,z)`	$\rightarrow$	Need `x`
`f({true},y,z)`	$\rightarrow$	Need `y`
`f({true},{5,6},z)`	$\rightarrow$	`{5,6}`

In subsequent evaluations of f, examining the cache will tell us to immediately evaluate the support of x, and if the support of x is `{true}`, we will immediately get the support of y, without any speculative computation required. If the support of y then turns out to be `{5,6}`, the result will be retrieved from the cache without any evaluation.

7.2.3 Support Computation

Aside from issues of laziness and memoization, the support computation is fairly straightforward, with the support of an expression being computed from the support of its subexpressions and its free variables. For example, to compute the support of `dist` $[e_1, ..., e_n]$, simply take the union of the supports of each of the e_i.

Some care is taken to use the supports of some subexpressions to simplify the computation of other subexpressions, so as to achieve the **sixth desideratum of exploiting supports**. The most basic manifestation of this idea is the application expression $e_1 e_2$, where we have *functional uncertainty*, i.e., uncertainty over the identity of the function to apply. For such an expression, IBAL first computes the support of e_1 to see which functions can be applied. Then, for each value f in the support of e_1, IBAL computes the support of applying f to e_2. Finally, the union of all these supports is returned as the support of $e_1 e_2$. For another example, consider an expression e of the form `if` e_1 `else` e_2 `then` e_3. A naive implementation would set the support of e to be the union of the supports of e_2 and e_3. IBAL is smarter, and performs a form of short-circuiting: if `true` is not in the support of e_1, the support of e_2 is not included in the support of e, and similarly for `false` and e_3.

7.3 Second Phase

In the second phase, the computation graph is solved from the bottom up. The solution for each node is generally not represented directly. Rather, it is represented as a set of *factors*. A factor *mentions* a set of variables, and defines a function from the values of those variables to real numbers. The variables mentioned by the factors in a solution include a special variable $*$ (pronounced "star") corresponding to the value of the expression, the free variables $\mathbf{X}$ of the expression, and other variables $\mathbf{Y}$. The solution specified by a set of factors $f_1, ..., f_n$ is $P(*|\mathbf{x}) = \frac{1}{\mathbf{Z}} \sum_{\mathbf{y}} \prod_{\mathbf{i}} \mathbf{f_i}(*, \mathbf{x}, \mathbf{y})$, where Z is a normalizing factor.[2] The set

2. The f_i do not need to mention the same variables. The notation $f_i(*, \mathbf{x}, \mathbf{y})$ denotes the value of f_i when $*$, $\mathbf{x}$ and $\mathbf{y}$ are projected onto the variables mentioned by f_i.

of factors at any node are a compact, implicit representation of the solution at that node. It is up to the solution algorithm to decide which $\mathbf{Y}$ variables to keep around, and which to eliminate.

At various points in the computation, the algorithm eliminates some of the intermediate variables $\mathbf{Y}$, using variable elimination (VE) (Dechter, 1996) to produce a new set of factors over the remaining variables. The root of the computation graph corresponds to the user's query. At the root there are no free variables. To compute the final answer, all variables other than $*$ are eliminated using VE, all remaining factors are multiplied together, and the result is normalized. By using VE for the actual process of computing probabilities, the algorithm achieves the **first desideratum of exploiting independence**. The main point is that unlike other programming-language based approaches, IBAL does not try to compute probabilities directly by working with a program, but rather converts a program into the more manipulable form of factors, and rests on tried and true technology for working with them.

In addition, this inference framework provides an easy method to satisfy the **third desideratum of exploiting high-level structure** of programs. As discussed in Section **??**, high-level structure is represented in IBAL using functions. In particular, the internals of a function are encapsulated inside the function, and are conditionally independent of the external world given the function inputs and outputs. From the point of view of VE, this means that we can safely eliminate all variables internal to the function consecutively. This idea is implemented by using VE to eliminate all variables internal to a function at the time the solution to the function is computed.

7.3.1 MICROFACTORS

Most implementations of VE in BNs represent a factor as a table. A table consists of a sequence of rows, each row consisting of a complete assignment of values to the factor variables and a real number. This representation is incapable of capturing low-level structure, and it also does not closely match the form of IBAL programs. Therefore, in order to achieve the **second desideratum of exploiting low-level structure**, IBAL uses a more refined representation called *microfactors*. Microfactors have similarities to other representations used for exploiting low-level structure, such as partial functions (Poole & Zhang, 2003) and algebraic decision diagrams (Bahar, Frohm, Gaona, Hachtel, Macii, Pardo, & Somenzi, 1993), but they were developed to match the structure of IBAL programs as closely as possible.

The design of microfactors is motivated by several observations about IBAL programs. First, it is common for values of variables to map to zero. Consider a comparison $e_1 == e_2$. A microfactor is created mentioning variables Y_1 and Y_2 for the outcomes of e_1 and e_2, and $*$, the outcome of the expression. The only assignments that have positive probability are those where Y_1 and Y_2 are equal and $*$ is `true`, or Y_1 and Y_2 are unequal and $*$ is false. All others are zero. To take advantage of the common zeros, only positive cases are represented explicitly in a microfactor.

The second observation is that we often don't care about the value of a variable, as in the case of context-specific independence. For example, given the expression `if x then y else z`, we will not care about `z` if `x` is `true`. Similarly, a factor will often have the same value for all

but a few values of a variable. Consider the expression `if x = 'a then y else z`. When translated into a factor, we obtain a function that is the same for all values of `x` except `a`. To take advantage of these cases, a row in a microfactor allows a variable to take on one of a set of values. Sets of values are represented as either V or $\overline{V}$, where $V = \{v_1, \ldots, v_n\}$ is an explicitly enumerated set of elements. The notation $\overline{V}$ denotes the complement of V, with respect to the universe of possible values. These are called *Zariski sets*, after the Zariski topology in real analysis. We can use $\overline{\emptyset}$ to denote the situation where something holds for all values of a variable, and $\overline{\{v\}}$ when something holds for all but the one value v. Next, a row over a set of variables $X_1, \ldots, X_n$ associates each variable X_i with a Zariski set Z_i, notated $< X_1 : Z_1, \ldots, X_n : Z_n >$. A row represents the set of tuples $< X_1 = x_1, \ldots, X_n = x_n >$ such that $x_i \in Z_i$. A row is empty if any of the Z_i is $\emptyset$. A microfactor is a sequence of disjoint, but not necessarily covering rows, where each row is associated with a real number.

Zariski sets have a pretty algebra:

$$\begin{array}{lll} V_1 \cup V_2 = V_1 \cup V_2 & V_1 \cap V_2 = V_1 \cap V_2 & V_1 - V_2 = V_1 - V_2 \\ V_1 \cup \overline{V_2} = \overline{V_2 - V_1} & V_1 \cap \overline{V_2} = V_1 - V_2 & V_1 - \overline{V_2} = V_1 \cap V_2 \\ \overline{V_1} \cup V_2 = \overline{V_1 - V_2} & \overline{V_1} \cap V_2 = V_2 - V_1 & \overline{V_1} - V_2 = \overline{V_1 \cup V_2} \\ \overline{V_1} \cup \overline{V_2} = \overline{V_1 \cap V_2} & \overline{V_1} \cap \overline{V_2} = \overline{V_1 \cup V_2} & \overline{V_1} - \overline{V_2} = V_2 - V_1 \end{array}$$

In order to implement VE, we need to define sum and product operations on microfactors. These in turn require intersection and difference operations on rows. Difference is defined in terms of intersection and complements. Intersection is straightforward:

$$< X_1 : Z_1^1, \ldots, X_n : Z_n^1 > \cap < X_1 : Z_1^2, \ldots, X_n : Z_n^2 > = < X_1 : Z_1^1 \cap Z_1^2, \ldots, X_n : Z_n^1 \cap Z_n^2 >$$

Complement is more complex, and defined recursively. Because rows are not closed under complement, the complement operation returns a set of rows, whose union is the complement of the given row. It is important for later that these rows are guaranteed to be disjoint. This is achieved by reasoning as follows: if a point $< x_1, \ldots, x_n >$ is in the complement of $< X_1 : Z_1, \ldots, X_n : Z_n >$, two disjoint cases are possible. Either $x_1 \notin Z_1$, in which case the rest of the x_i can be anything, or else $x_1 \in Z_1$, but $< x_2, \ldots, x_n >$ is in the complement of $< X_2 : Z_2, \ldots, X_n : Z_n >$. We can achieve the right effect by recursively computing the latter complement, which returns a list of rows, extending each row with the condition $X_1 : Z_1$, and adding the row representing the first case. The full formulas for complement are[3]

$$\begin{array}{ll} \text{Base case} & \neg <> = \{\} \\ \text{Inductive case} & \neg < X_1 : Z_1, \ldots, X_n : Z_n > = \\ & < X_1 : \neg Z_1, X_2 : \overline{\emptyset}, \ldots, X_n : \overline{\emptyset} > \cup \\ & \{< X_1 : Z_1, X_2 : W_2, \ldots, X_n : W_n > \mid \begin{array}{l} < X_2 : W_2, \ldots, X_n : W_n > \in \\ \neg < X_2 : Z_2, \ldots, X_n : Z_n > \end{array} \} \end{array}$$

The following assumptions about factors are maintained throughout the variable elimination process: (1) A factor contains no empty rows; (2) None of the rows are associated with the number zero; (3) The rows are pairwise disjoint. A factor represents a function from the values of the variables to real numbers. By assumption 3, any tuple of values of

3. A detail is that the first case can hold only if Z_1 is not $\overline{\emptyset}$. When Z_1 is $\overline{\emptyset}$, the set for the first case is dropped from the result, so that no empty sets are included in the list.

the variables can be contained in at most one row. If it is contained in a row, the value associated with the tuple is the row value. If the tuple is not contained in a row, the value associated with it is zero.

To implement variable elimination, we need multiplication and summation operators on factors. Multiplication is straightforward. To multiply two factors f_1 and f_2, we take the cross product of all rows in f_1 and f_2. For each pair of rows r_1 in f_1 and r_2 in f_2,

1. Let p be the product of the real numbers associated with r_1 and r_2. p is non-zero by assumption.
2. Extend r_1 so that it covers all variables mentioned by f_2, by adding a constraint $X : \overline{\emptyset}$ for variables X mentioned by f_2 but not f_1.
3. Similarly extend r_2 to cover all variables mentioned by f_1.
4. Let r be the intersection of r_1 and r_2.
5. If r is empty, throw it out, otherwise add a row associating r with p in the result.

In the actual implementation, a precomputation is performed to calculate the indices of every result variable in r_1 and r_2. As a result, steps 2 and 3 do not need to be physically performed, and the intersections can be performed very quickly.

The summation operation is more complex. First, consider what happens in ordinary VE when we sum over a variable. We simply delete that variable from each row. After that, two rows are either equal or disjoint. If they are equal, their associated numbers can simply be added. This property holds because every row represents a single point. It no longer holds when rows represent sets of points, even though the original rows before deleting the variable were pairwise disjoint. Consider the following example:

X	Y	
{a}	{r}	0.1
{b}	{s}	0.3
$\overline{\emptyset}$	{t}	0.6

These rows are all pairwise disjoint, because they all specify a different value for variable Y. However, consider what happens when we sum over Y. After deleting Y, we have the three rows $< X : \{a\} >$, $< X : \{b\} >$ and $< X : \overline{\emptyset} >$. $\overline{\emptyset}$ overlaps with, but is not equal to, $\{a\}$ and $\{b\}$. We cannot simply leave these three rows in the answer, because they do not satisfy the pairwise disjointness property, nor can we simply add them. The answer we want is

X	
{a}	0.7
{b}	0.9
$\overline{\{a, b\}}$	0.6

which correctly represents the summation over Y of the sum represented by the first factor.

In order to perform summation correctly, the following iterative process is used:

1. Let R be the set of rows after deleting the summed over variable.
2. While the rows in R are not pairwise disjoint:
   (a) Remove two rows r_1 and r_2 that have a non-empty intersection from R.
   (b) Let p_1 and p_2 be the probabilities associated with r_1 and r_2.
   (c) Compute $r_1 \cap r_2$ and associate it with $p_1 + p_2$ in R.
   (d) Compute $r_1 - r_2$ (recall that it is a set of rows) and associate each member of the set with p_1 in R.
   (e) Compute $r_2 - r_1$ and associate each member with p_2 in R.
3. Return R.

In practice the implementation examines rows in a more orderly fashion than that indicated by the above algorithm. However, the above process is the essence of the operation. It is guaranteed to terminate, because the rows represent smaller and smaller sets after each step. In the limit, each row will represent a single point as in the tabular representation, in which case the process will certainly stop.

It is important to note that the operations for multiplication and summing over a variable provided here make no assumptions about the representation of rows, other than that there are intersection and difference operations defined on them. Thus they can be used for any implementation of VE that uses a more refined representation than tables for the factors.

Now that these operations have been defined, VE can be performed in the usual manner. An interesting research question is to study the complexity of VE in light of the new representations used for factors. Concepts such as the width of the induced graph may no longer be sufficient to characterize the cost of inference, because there may be other properties of a particular problem that cause it to be more efficient than indicated. Put differently, when compact representations of factors are available, the size of factors is not necessarily exponential in the number of variables they mention. From a practical point of view, it would be interesting to study whether the same algorithms for choosing variable elimination orderings work when microfactors are used.

7.3.2 Translating Programs Into Microfactors

The next step in IBAL inference is to translate a program into a set of micro-structural factors, and then perform VE. The goal is to produce factors that capture all the structure in the program, including both the independence structure and the low-level structure.

The translation is expressed through a set of rules, each of which takes an expression of a certain form and returns a set of factors. The notation $T[e]$ is used to denote the translation rule for expression e. Thus, for a constant expression $'v$ the rule is[4]

$$T['v] = \begin{array}{|c|c|}\hline * & \\ \hline v & 1 \\ \hline \end{array}$$

The Boolean constants and `lambda` and `fix` expressions are treated similary.

4. For convenience, I omit the set brackets for singletons.

For a variable expression, $T[x]$, we need to make sure that the result has the same value as x. If x is a simple variable, whose values are symbols, the rule is as follows. Assuming $v_1, \ldots, v_n$ are the values in the support of x, this is achieved with the rule

$$T[x] = \begin{array}{|c|c|c|} \hline * & x & \\ \hline v_1 & v_1 & 1 \\ & \cdots & \\ v_n & v_n & 1 \\ \hline \end{array}$$

Here, we exploit the fact that an assignment of values to variables not covered by any row has value 0.

If x is a complex variable with multiple fields, each of which is itself complex, we could use the above rule, considering all values in the cross-product space of the fields of x. However, that is unnecessarily inefficient. Rather, for each field a of x, we ensure separately that $*.a$ is equal to $x.a$. If a itself is complex, we break that equality up into fields. We end up with a factor like the one above for each simple chain c defined on x. If we let the simple chains be $c^1, \ldots, c^m$, and the possible values of c^i be $v^i_1, \ldots, v^i_{n^i}$, we get the rule

$$T[x] = \bigcup_{i=1}^{m} \begin{array}{|c|c|c|} \hline *.c^i & x.c^i & \\ \hline v^1_i & v^1_i & 1 \\ & \cdots & \\ v^1_{n^i} & v^1_{n^i} & 1 \\ \hline \end{array}$$

The total number of rows according to this method is $\sum_{i=1}^{m} n^i$, rather than $\prod_{i=1}^{m} n^i$ for the product method.

Next we turn to variable definitions. Recall that those are specified in IBAL through a let expression of the form `let` $x = e_1$ `in` e_2. We need some notation: if $\mathbf{F}$ is a set of factors, $\mathbf{F}^{\mathbf{c^1}}_{\mathbf{c^2}}$ denotes the same set as $\mathbf{F}$, except that chain c^1 is substituted for c^2 in all the factors in $\mathbf{F}$. Now the rule for `let` is simple. We compute the factors for e_1, and replace $*$ with x. We then conjoin the factors for e_2, with no additional change. The full rule is[5] $T[\texttt{let } x = e_1 \texttt{ in } e_2] = T[e_1]^x_* \cup T[e_2]$.

For `if-then-else` expressions, we proceed as follows. First we define a primitive `prim_if` (x, y, z) that is the same as if but only operates on variables. Then we can rewrite

$$\begin{array}{l} \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 = \\ \texttt{let } x = e_1 \texttt{ in} \\ \texttt{let } y = e_2 \texttt{ in} \\ \texttt{let } z = e_3 \texttt{ in} \\ \texttt{prim\_if } (x, y, z) \end{array}$$

Now, all we need is a translation rule for `prim_if` and we can invoke the above `let` rule to translate all `if` expressions.[6] Let the simple chains on y and z be $c^1, \ldots, c^m$. (They must have the same set of simple chains for the program to be well typed.) Using the same

5. A fresh variable name is provided for the bound variable to avoid name clashes.
6. In practice, if e_1, e_2 or e_3 are already variable expressions, we can omit the let expression defining x, y or z and use them directly in the `prim_if`.

notation as before for the possible values of these chains, a naive rule for `prim_if` is as follows:

$$T[\texttt{prim\_if}(x,y,z)] = \bigcup_{i=1}^{m}$$

$*.c^i$	x	$y.c^i$	$z.c^i$	
v_1^i	T	v_1^i	$\overline{\emptyset}$	1
		$\cdots$		
$v_{n^i}^i$	T	$v_{n^i}^i$	$\overline{\emptyset}$	1
v_1^i	F	$\emptyset$	v_1^i	1
		$\cdots$		
$v_{n^i}^i$	F	$\overline{\emptyset}$	$v_{n^i}^i$	1

This rule exploits the context-specific independence present in any `if` expression: the outcome is independent of either the `then` clause or the `else` clause given the value of the test. The CSI is captured in the $\overline{\emptyset}$ entries for the irrelevant variables. However we can do even better. This rule unites $y.c^i$ and $z.c^i$ in a single factor. However, there is no row in which both are simultaneously relevant. We see that `if` expressions satisfy a stronger property than CSI. To exploit this property, the `prim_if` rule produces two factors for each c^i whose product is equal to the factor above.

$$T[\texttt{prim\_if}(x,y,z)] =$$

$$\bigcup_{i=1}^{m} \left\{ \right.$$

$*.c^i$	x	$y.c^i$	
v_1^i	T	v_1^i	1
	$\cdots$		
$v_{\underline{n}^i}^i$	T	$v_{\underline{n}^i}^i$	1
$\emptyset$	F	$\emptyset$	1

,

$*.c^i$	x	$z.c^i$	
v_1^i	F	v_1^i	1
	$\cdots$		
$v_{\underline{n}^i}^i$	F	$v_{\underline{n}^i}^i$	1
$\emptyset$	T	$\emptyset$	1

$\left. \right\}$

Note the last row in each of these factors. It is a way of indicating that the factor is only relevant if x has the appropriate value. For the first factor, if x has the value F, the factor has value 1 whatever the values of the other variables, and similarly for the other factor. The number of rows in the factors for c^i is two more than for the previous method, because of the "irrelevance rows". However, we have gained in that $y.c^i$ and $z.c^i$ are no longer in the same factor. Considering all the c^i, the moral graph for the second approach contains m fewer edges than for the first approach. Essentially, the variable x is playing the role of a separator for all the pairs $y.c^i$ and $z.c^i$. If we can avoid eliminating x until as late as possible, we may never have to connect many of the $y.c^i$ and $z.c^i$.

None of the expression forms introduced so far contained uncertainty. Therefore, every factor represented a zero-one function, in other words, a constraint on the values of variables. Intermediate probabilities are finally introduced by the dist expression, which has the form `dist` $[p_1 : e_1, \ldots, p_n : e_n]$. As in the case of `if`, we introduce a primitive `prim_dist` $(p_1, \ldots, p_n)$, which selects an integer from 1 to n with the corresponding probability. We also use `prim_case` which generalizes the `prim_if` above to take an integer test

with n possible outcomes. We can then rewrite

$$\begin{array}{l}\texttt{dist}\ [p_1 : e_1, \ldots, p_n : e_n] = \\ \texttt{let}\ x_1 = e_1\ \texttt{in} \\ \ldots \\ \texttt{let}\ x_n = e_n\ \texttt{in} \\ \texttt{let}\ z = \ \texttt{prim\_dist}\ (p_1, \ldots, p_n)\ \texttt{in} \\ \texttt{prim\_case}\ (z, [x_1, \ldots, x_n])\end{array}$$

To complete the specification, we only need to provide rules for `prim_dist` and `prim_case`. The `prim_dist` rule is extremely simple:

$$T[\texttt{prim\_dist}(p_1, \ldots, p_n)] = \begin{array}{|c|c|}\hline * & \\ \hline 1 & p_1 \\ & \ldots \\ n & p_n \\ \hline \end{array}$$

The `prim_case` rule generalizes the rule for `prim_if` above. It exploits the property that no two of the x_j can be relevant, because the `dist` expression selects only one of them. This technique really comes into its own here. If there are m different chains defined on the result, as before, and n different possible outcomes of the `dist` expression, the number of edges removed from the moral graph is $m * n$. The rule is

$$T[\texttt{prim\_case}(z, [x_1, \ldots, x_n])] =$$

$$\bigcup_{i=1}^{m} \bigcup_{j=1}^{n} \begin{array}{|c|c|c|c|}\hline *.c^i & z & x_j.c^i & \\ v_1^i & j & v_1^i & 1 \\ & \ldots & & \\ v_{\underline{n}^i}^i & j & v_{\underline{n}^i}^i & 1 \\ \emptyset & \{j\} & \emptyset & 1 \\ \hline \end{array}$$

The rules for record construction and field access expressions are relatively simple, and are omitted.

Next, we turn to the mechanism for applying functions. It also needs to be able to handle *functional uncertainty* — the fact that the function to be applied is itself defined by an expression, over whose value we have uncertainty. To start with, however, let us assume that we know which particular function we are applying to a certain set of arguments. For a function f, let $f.x_1, \ldots, f.x_n$ denote its formal arguments, and $f.b$ denote its body. Let $A[f, e_1, \ldots, e_n]$ denote the application of f to arguments defined by expressions $e_1, \ldots, e_n$. Then

$$A[f, e_1, \ldots, e_n] = T \left[\begin{array}{l} \texttt{let} f.x_1 = e_1 \texttt{in} \\ \ldots \\ \texttt{let} f.x_n = e_n \texttt{in} \\ f.b \end{array} \right]$$

By the `let` rule presented earlier, this will convert $f.b$ into a set of factors that mention the result variable $*$, the arguments $f.x_i$, and variables internal to the body of f. Meanwhile,

each of the e_i is converted into a set of factors defining the distribution over $f.x_i$.

$$A[f, e_1, \ldots, e_n] = T[f.b] \cup \bigcup_i T[e_i]^{f.x_i}_*$$

To exploit encapsulation, we want to eliminate all the variables that are internal to the function call before passing the set of factors out to the next level. This can be achieved simply by eliminating all temporary variables except for those representing the $f.x_i$ from $T[f.b]$. Thus, a VE process is performed for every function application. The result of performing VE is a conditional distribution over $*$ given the $f.x_i$.[7]

Normally in VE, once all the designated variables have been eliminated, the remaining factors are multiplied together to obtain a distribution over the uneliminated variables. Here that is not necessary: performing VE returns a set of factors over the uneliminated variables, that is passed to the next level up in the computation. Delaying the multiplication can remove some edges from the moral graph at the next level up.

Now suppose we have an application expression $e_0(e_1, \ldots, e_n)$. The expression e_0 does not have to name a particular function, and there may be uncertainty as to its value. We need to consider all possible values of the function, and apply each of those to the arguments. Let F denote the support of e_0. Then for each $f_i \in F$, we need to compute $A_i = A[f_i, e_1, \ldots, e_n]$ as above.

Now, we cannot simply take the union of the A_i as part of the application result, since we do not want to multiply factors in different A_i together. The different A_i represent the conditional distribution over the result for different function bodies. We therefore need to condition A_i on F being f_i. This effect is achieved as follows. Let $A_i^1, \ldots, A_i^m$ be the factors in A_i, and let $(r_1^j, p_1^j), \ldots, (r_{\ell j}^j, p_{\ell j}^j)$ be the rows in factor A_i^j. Then we can write

$B_i = \bigcup_{j=1}^{m}$

F	$*, f_i.x_1, \ldots, f_i.x_n$	
f_i	r_1^j	p_1^j
	$\ldots$	
f_i	$r_{\ell j}^j$	$p_{\ell j}^j$
$\overline{\{f_i\}}$	$\overline{\emptyset}$ for all	1

In words, each B_i^j is formed from the corresponding A_i^j in two steps. First, A_i^j is extended by adding a column for F, and setting its value to be equal to f_i. The effect is to say that when F is equal to f_i, we want A_i^j to hold. Then, a row is added saying that when F is unequal to f_i, the other variables can take on any value and the result will be 1. The effect is to say that A_i^j does not matter when $F \neq f_i$. We can now take the union of all the B_i. To complete the translation rule for function application, we just have to supply the distribution over F:

$$T[e_0(e_1, \ldots, e_n)] = \cup_i B_i \cup T[e_0]^F_*$$

7. There may also be variables that are free in the body of f and not bound by function arguments. These should also not be eliminated.

7.4 Learning and Decision Making

IBAL's inference algorithms for learning and decision making are completely standard, and use the previously described probabilistic inference algorithm as a subroutine. Learning is based on the EM algorithm (Dempster, Laird, & Rubin, 1977). The inference algorithm is used for the E step.

Decision making uses backward induction (Shachter, 1986). It solves each of the decisions within a function in turn, from the last one backwards. For each decision, it considers all possible values of the informational parents, and all possible values of the decision variable. For each set of values of the informational parents, and each value of the decision variable, it determines the expected utility achieved under the distribution over environments conditioned on the given values. It then chooses the value of the decision variable that maximizes this expected utility, as a function of the informational parents.

Nested function calls are treated as fully encapsulated black boxes, to be solved recursively. In solving an MDP, it treats the future as a function call which is then solved recursively. As always, the arguments to the function call are memoized, so if the function is applied twice to the same state it is only evaluated once. In this way, dynamic programming is achieved. In fact, the IBAL inference algorithm performs the same computations as the value iteration algorithm for MDPs.

8. Lessons Learned and Conclusion

The IBAL implementation represents the culmination of several years of investigation, that begin with the original "stochastic Lisp" paper (Koller et al., 1997) and continued with the SPOOK system (Pfeffer et al., 1999). A number of important lessons were learned from the process:

- Stochastic programming languages are surprisingly complex, and a sophisticated algorithm such as the one in this paper is needed to implement them.
- As a corollary, a single mechanism is unlikely to achieve all the goals of inference in a complex system. The move to the two-phase approach greatly simplified the implementation, but was also an admission that the implementation had entered a new level of complexity.
- The design of the language and of the inference algorithm go hand in hand. The set of language constructs in IBAL was chosen to support the specific inference goals described in this paper.
- Different approaches that are individually inadequate may each have something to contribute to the overall solution. Programming language evaluation approaches provide a natural way to work with programs, and were used in constructing the computation graph. SPOOK's approach of using local VE processes for different model components was used. Also, the KBMC approach of separating the model analysis and probability computation components was used, albeit in a very different way.
- Beware unexpected interactions between goals! (Koller et al., 1997) blithely declared that lazy evaluation and memoization would be used. In retrospect, combining the two mechanisms was the single most difficult thing in the implementation.

This paper has presented the probabilistic inference mechanism for IBAL, a highly expressive probabilistic representation language. A number of apparently conflicting desiderata for inference were presented, and it was shown how IBAL's inference algorithm satisfies all of them. It is hoped that the development of IBAL provides a service to the community in two ways. First, it provides a blueprint for anyone who wants to build a first-order probabilistic reasoning system. Second, and more important, it is a general-purpose system that has been released for public use. In future it will hopefully be unnecessary for designers of expressive models to have to build their own inference engine. IBAL has succesfully been tried on BNs, HMMs (including infinite state-space models), stochastic grammars and probabilistic relational models. IBAL has also been used successfully as a teaching tool in a probabilistic reasoning course at Harvard. Its implementation consists of approximately 10,000 lines of code. It includes over 50 test examples, all of which the inference engine is able to handle. IBAL's tutorial and reference manuals are both over 20 pages long.

Of course, there are many models for which the techniques presented in this paper will be insufficient, and for which approximate inference is needed. The next step of IBAL development is to provide approximate inference algorithms. IBAL's inference mechanism already provides one way to do this. One can simply plug in any standard BN approximate inference algorithm in place of VE whenever a set of factors has to be simplified. However, other methods such as Markov chain Monte Carlo will change the way programs are evaluated, and will require a completely different approach.

References

Bahar, R. I., Frohm, E. A., Gaona, C. M., Hachtel, G. D., Macii, E., Pardo, A., & Somenzi, F. (1993). Algebraic decision diagrams and their applications. In *IEEE/ACM International Conference on CAD.*

Bellman, R. (1957). *Dynamic Programming.* Princeton Press.

Dean, T., & Kanazawa, K. (1989). A model for reasoning about persistence and causation. *Computational Intelligence*, *5*, 142–150.

Dechter, R. (1996). Bucket elimination : a unifying framework for probabilistic inference. In *Uncertainty in Artificial Intelligence (UAI).*

Dempster, A. P., Laird, N. M., & Rubin, D. B. (1977). Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society, B*, *39*(1), 1–38.

Heckerman, D., & Breese, J. S. (1994). A new look at causal independence. In *Uncertainty in Artificial Intelligence (UAI).*

Howard, R. A., & Matheson, J. E. (1984). Influence diagrams. In *Readings on the Principles and Applications of Decision Analysis, 2*, pp. 719–762. Strategic Decisions Group, Menlo Park, CA.

Kersting, K., & de Raedt, L. (2000). Bayesian logic programs. In *Proceedings of the Work-In-Progress Track at the 10th International Conference on Inductive Logic Programming.*

Koller, D., McAllester, D., & Pfeffer, A. (1997). Effective Bayesian inference for stochastic programs. In *National Conference on Artificial Intelligence (AAAI).*

Koller, D., & Pfeffer, A. (1997). Object-oriented Bayesian networks. In *Uncertainty in Artificial Intelligence (UAI)*.

Koller, D., & Pfeffer, A. (1998). Probabilistic frame-based systems. In *National Conference on Artificial Intelligence (AAAI)*.

Koller, D., & Pfeffer, A. (2000). Semantics and inference for recursive probability models. In *National Conference on Artificial Intelligence (AAAI)*.

Laskey, K. B., & Mahoney, S. M. (1997). Network fragments: Representing knowledge for constructing probabilistic models. In *Uncertainty in Artificial Intelligence (UAI)*.

Li, Z., & D'Ambrosio, B. (1994). Efficient inference in bayes' networks as a combinatorial optimization problem. *International Journal of Approximate Inference, 11*.

Muggleton, S. (2001). Stochastic logic programs. *Journal of Logic Programming*. Accepted subject to revision.

Ngo, L., & Haddawy, P. (1996). Answering queries from context-sensitive probabilistic knowledge bases. *Theoretical Computer Science*.

Pasula, H., & Russell, S. (2001). Approximate inference for first-order probabilistic languages. In *International Joint Conference on Artificial Intelligence (IJCAI)*.

Pfeffer, A. (2000). *Probabilistic Reasoning for Complex Systems*. Ph.D. thesis, Stanford Univeristy.

Pfeffer, A. (2004). Repeated observation models. In *National Conference on Artificial Intelligence (AAAI)*.

Pfeffer, A., Koller, D., Milch, B., & Takusagawa, K. T. (1999). SPOOK: A system for probabilistic object-oriented knowledge representation. In *Uncertainty in Artificial Intelligence (UAI)*.

Pless, D., & Luger, G. (2001). Toward general analysis of recursive probability models. In *Uncertainty in Artificial Intelligence (UAI)*.

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

Poole, D., & Zhang, N. L. (2003). Exploiting contextual independence in probabilistic inference. *Journal of Artificial Intelligence Research (JAIR)*.

Sanghai, S., Domingos, P., & Weld, D. (2003). Dynamic probabilistic relational models. In *International Joint Conference on Artificial Intelligence (IJCAI)*.

Shachter, R. D. (1986). Evaluating influence diagrams. *Operations Research, 34*, 871–882.

Spiegelhalter, D. J., Thomas, A., Best, N., & Gilks, W. R. (1995). BUGS 0.5 : Bayesian inference using Gibbs sampling manual. Tech. rep., Institute of Public Health, Cambridge University.