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

Avi Pfeffer

This chapter 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 chapter 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.

14.1 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 toward 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 chapter presents IBAL, a probabilistic rational programming language. IBAL, pronounced "eyeball," stands for *I*ntegrated *B*ayesian *A*gent *L*anguage. As its name suggests,

it integrates various aspects of probability-based rational behavior, including probabilistic reasoning, Bayesian parameter estimation, and decision-theoretic utility maximization. This chapter will focus on the probabilistic representation and reasoning capabilities of IBAL, and not discuss the learning and decision-making aspects.

High-level probabilistic languages have generally fallen into two categories. The first category is rule-based [19, 13, 5]. 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 (BN) can be constructed describing a joint distribution over all the first-order variables in the domain.

The second category of language is object-based [7, 8, 10]. 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 chapter 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 spe-

cial 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 chapter begins by presenting the IBAL language. The initial focus is on the features that allow description of generative probabilistic models. After presenting examples, the chapter presents the declarative semantics of IBAL.

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 BN is encoded in IBAL, the IBAL inference algorithm should perform the same operations as a BN inference algorithm. This chapter describes the IBAL inference algorithm and shows how it generalizes many existing frameworks, including Bayesian networks, hidden Markov models (HMMs), and stochastic context free grammars (SCFGs). Seven desiderata for a general-purpose inference algorithm are presented, and it is shown how IBAL's algorithm satisfies all of them simultaneously.

14.2 The IBAL Language

IBAL is a rich language. We first describe the core of the language which is used to build generative probabilistic models. Then we discuss how to encode observations in models. Finally we present some syntactic sugar that makes the language easier and more natural to use.

14.2.1 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 14.4. 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 ith 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.

Comparison The expression $e_1 == e_2$ corresponds to the experiment in which e_1 and e_2 are evaluated. The result is `true` if the values of e_1 and e_2 are the same; otherwise it is `false`.

Example 14.1
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 14.2
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 = lambda () -> dist [ 0.5 : 'heads, 0.5 : 'tails ] in
let biased = lambda () -> dist [ 0.9 : 'heads, 0.1 : 'tails ] in
let pick = lambda () -> dist [ 0.5 : fair, 0.5 : biased ] in
let c = pick () in
let x = c () in
let y = c () in
<x:x, y:y> \ \ \bbox
```

14.2.2 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 BNs, probabilistic relational models, HMMs, dynamic Bayesian networks, and SCFGs. 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 14.3
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`. ∎

14.2.3 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.

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` $f(x_1, \ldots, x_n) = e$.

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 *definitions*. A *block* is a piece of IBAL code consisting of a sequence of variable definitions.

Example 14.4
For example, we can rewrite our coins example using definitions.

```
fair() = dist [ 0.5 : 'heads, 0.5 : 'tails ]
biased()  = dist [ 0.9 : 'heads, 0.1 : 'tails ]
pick() = dist [ 0.5 : fair, 0.5 : biased ]
c = pick()
x = c()
y = c()
```

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` $[\alpha : \texttt{true}, 1-\alpha : \texttt{false}]$. 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 BNs. 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 14.5

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 nonterminal 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))
```

∎

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 system is based on that of ML. 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.

14.3 Examples

Example 14.6

Encoding a BN is easy and natural in IBAL. We include a definition for each variable in the network. A `case` expression 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 others. 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 14.7

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 : lambda () -> 0,
       trans : lambda (n) -> dist [ 0.5 : n++, 0.5 : n-- ] >;
```

It is easy to see how to generalize this example to HMMs by providing an observation function, and then specifying the observations using `obs` expressions. Then, combined with the previous example of BNs, we can generalize this to dynamic Bayesian networks [2]. ∎

Example 14.8

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 [17], a good deal of code was devoted to handling number uncertainty. In this example, we 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 14.9

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* [15]. A repetition model is used to describe a sequence of elements in which repetition of elements from earlier in the sequence is a common occurrence. 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)
```

Similarly to the way we defined Markov models earlier, a repetition HMM takes `init`, `trans`, and `obs` functions as arguments. The parameter ρ must be supplied. If we used all of IBAL's features it could be a learnable parameter. In our example

we set it to 0.1. 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 `dist` 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())
```

∎

In addition to these examples, IBAL can represent PRMs, and by extension dynamic PRMs [21]. Meanwhile, the decision-making constructs of IBAL allow the encoding of influence diagrams and Markov decision processes.

14.4 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 is distributional: the meaning of a program is specified in terms of a probability distribution over values.

14.4.1 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 probability distribution over 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 distribution over environments is Δ. We also use the notation $\mathcal{M}[e]\ \epsilon\ v$ to denote the probability of v under the meaning of e when the probability distribution over environments assigns positive probability only to ϵ.

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

$$\mathcal{M}[x]\ \Delta\ v = \sum_{\epsilon:\epsilon(x)=v} \Delta(\epsilon).$$

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.

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

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:

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

To define the meaning of an expression `let` $x = e_1$ `in` $e2$, we first define a probability distribution Δ' over extended environments that are produced by binding x with any possible value. The probability of an extended environment is the probability of the original environment that is being extended times the probability that e_1 will produce the given value in the original environment. We then define the meaning of the entire expression to be the meaning of e_2 under Δ'. The notation $\epsilon[x/v']$ indicates the environment produced by extending ϵ by binding x to v'.

$$\mathcal{M}[\texttt{let}\ x = e_1\ \texttt{in}\ e_2]\ \Delta\ v = \mathcal{M}[e_2]\ \Delta'\ v$$
$$\text{where } \Delta'(\epsilon') = \begin{cases} \Delta(\epsilon)(\mathcal{M}[e_1]\ \epsilon\ v') & \text{if } \epsilon' = \epsilon[x/v'] \\ 0 & \text{otherwise} \end{cases}$$

`lambda` and `fix` expressions are treated as constants whose values are closures. The only difference is that the closure specifies an environment, so we take the probability that the current environment is the closure environment.

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

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

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 each argument x_i to v_i.

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

The meaning of a tuple expression is given by

$$\begin{array}{c} \mathcal{M}[< x_1 : e_1, \ldots, x_n : e_n >]\ \Delta\ v = \\ \left\{ \begin{array}{ll} \sum_\epsilon \Delta(\epsilon) \prod_{i=1}^n \mathcal{M}[e_i]\ (\epsilon)\ v_i & \text{if } v =< x_1 : v_1, \ldots, x_n : v_n >, \\ 0 & \text{otherwise} \end{array} \right. \end{array}$$

The meaning of extracting a component from a tuple is

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

Finally, the probability of a comparison being true is derived by taking the sum, over all possible values, of the probability that both expressions produce the value.

$$\mathcal{M}[e_1 == e_2]\ \Delta`v = \begin{cases} p & \text{if } v = \texttt{true} \\ 1-p & \text{if } v = \texttt{false} \\ 0 & \text{otherwise} \end{cases}$$
$$\text{where } p = \sum_{\epsilon} \Delta(\epsilon) \sum_{v'} (\mathcal{M}[e_1]\ \epsilon\ v')(\mathcal{M}[e_2]\ \epsilon\ v')$$

The distributional semantics captures observations quite simply. The effect of an observation is to condition the distribution Δ over environments on the observation holding. When the probability that the observation holds is zero, the probability of the expression is defined to be zero.

$$\mathcal{M}[\texttt{obs}\ x = v'\ \texttt{in}\ e]\ \Delta\ v = \begin{cases} \frac{\sum_{\epsilon:\epsilon(x)=v'} \Delta(\epsilon)(\mathcal{M}[e]\ \epsilon\ v)}{P(x=v')} & \text{if } P(x=v') > 0 \\ 0 & \text{if } P(x=v') = 0 \end{cases}$$
$$\text{where } P(x=v') = \sum_{\epsilon:\epsilon(x)=v'} \Delta(\epsilon)$$

14.4.2 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, an SCFG 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 14.10
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`. ∎

The distributional semantics presented earlier is agnostic about whether it is eager or lazy. It simply presents a set of equations, and says nothing about how the equations are evaluated. Both eager and lazy interpretations are possible. The meaning of an expression under either interpretation is only well-defined when the process of evaluating it converges. The eager and lazy semantics do not necessarily agree. The eager semantics may diverge in some cases where the lazy semantics produces a result. However, if the eager semantics converges, the lazy semantics will produce the same result.

14.5 Desiderata for Inference

IBAL is able to capture many traditional kinds of representations, such as BNs, HMMs, and SCGGs. It can also express more recent models such as object-oriented Bayesian networks (OOBNs) and relational probability models [16]. 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 BNs, 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 [4, 20]. 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 [17] 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.

14.6 Related Approaches

Previous approaches to inference in high-level probabilistic languages have generally fallen into four categories. On one side are approaches that use approximate inference, particularly Markov chain Monte Carlo methods. This is the approach used in BUGS [23] and the approach taken by Pasula and Russell in their first-order probabilistic logic [14]. While exact inference may be intractable for many models, and approximate strategies are therefore needed, the goal of this chapter 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. [19, 13, 10, 5]). In this approach, a knowledge base describes the general probabilistic mechanisms. These are combined with ground facts to produce a BN 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.

Sato and Kameya [22] present a more advanced version of this approach that achieves some of the aims of this paper. They use a tabling procedure to avoid performing redundant computations. In addition, their approach is query-directed. However they do not exploit low-level independence or weak interaction between objects, nor do they utilize observations or support.

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 [9, 18] or logic programming [12]. 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. [9] 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 variable elimination (VE) using a particular elimination order: namely, from the last variable in the program upward. 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. [9] 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 [9]. However, this catchall term fails to capture the distinctions between the different ways knowledge can be exploited. A careful implementation of the algorithm in [9] 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 [17, 16], a BN 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 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 [6] was provided for dealing with infinitely recursive models, but it is not an approximate inference algorithm.

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 [17], 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 [17], 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 14.8, so it would be highly beneficial if IBAL's inference algorithm can exploit it automatically.

14.7 Inference

14.7.1 Inference Overview

If we examine the desiderata of section 14.5, 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 symbolic probabilistic inference (SPI) algorithm for BN inference [11], 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 piggybacking 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 occurring in the solution. By contrast, IBAL's computation graph contains a node for every *distinct*

1. Simon Peyton-Jones, personal communication.

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.

14.7.2 First Phase

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. IBAL's observation propagation process is sound but not complete. For an SCFG, it is able to infer when the output string is finite that only a finite computation is needed to produce it. The details are omitted here.

14.7.2.1 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.

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.

14.7.2.2 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.

14.7.3 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 $P(*|\mathbf{x}) = \frac{1}{Z} \sum_{\mathbf{y}} \prod_i f_i(*, \mathbf{x}, \mathbf{y})$, where Z is a normalizing factor.[2] The set 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 VE [3] to produce a new set of factors over the remaining variables. The root of the computation graph corresponds to the user's

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.

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 the high-level structure** of programs. As discussed in section 14.5, 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.

14.7.3.1 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 [20] and algebraic decision diagrams [1], 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.

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 in turn is defined in terms of intersection and complements. Intersection is straightforward. 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. These rows are guaranteed to be disjoint. Details of the operation are omitted.

To implement VE, we need multiplication and summation operators on factors. Multiplication is straightforward. For summation, an iterative process is used that guarantees that the resulting microfactor correctly represents the sum over the given variable in the original factor, and that its rows are disjoint. Again details are omitted.

14.7.3.2 Translating Programs into Microfactors

The next step in IBAL inference is to translate a program into a set of microfactors, 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 microfactors. The notation $T[e]$ is used to denote the translation rule for expression e. Thus, for a constant expression $'v$ the rule is[3]

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

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

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

3. For convenience, we omit the set brackets for singletons.

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 $v_1^i, \ldots, v_{n^i}^i$, we get the rule

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

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 $\boldsymbol{F}$ is a set of factors, $\boldsymbol{F}_{c^2}^{c^1}$ denotes the same set as $\boldsymbol{F}$, except that chain c^1 is substituted for c^2 in all the factors in $\boldsymbol{F}$. 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[4] $T[\texttt{let}\ x = e_1\ \texttt{in}\ e_2] = T[e_1]_*^x \cup T[e_2]$.

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

$$\begin{aligned}
&\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 = \\
&\texttt{let}\ x = e_1\ \texttt{in} \\
&\texttt{let}\ y = e_2\ \texttt{in} \\
&\texttt{let}\ z = e_3\ \texttt{in} \\
&\texttt{prim_if}\ (x, y, z)
\end{aligned}$$

4. A fresh variable name is provided for the bound variable to avoid name clashes.

Now, all we need is a translation rule for `prim_if` and we can invoke the above `let` rule to translate all `if` expressions.[5] 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 notation as before for the possible values of these chains, a naive rule for `prim_if` is as follows:

$$T[\texttt{prim_if}(x,y,z)] = \bigcup_{i=1}^{m} \begin{array}{|c|c|c|c|c|} \hline *.c^i & x & y.c^i & z.c^i & \\ \hline v_1^i & T & v_1^i & \overline{\emptyset} & 1 \\ & & \cdots & & \\ v_{n^i}^i & T & v_{n^i}^i & \overline{\emptyset} & 1 \\ v_1^i & F & \overline{\emptyset} & v_1^i & 1 \\ & & \cdots & & \\ v_{n^i}^i & F & \overline{\emptyset} & v_{n^i}^i & 1 \\ \hline \end{array}$$

This rule exploits the context-specific independence (CSI) 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.

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

$$\bigcup_{i=1}^{m} \left\{ \begin{array}{|c|c|c|c|} \hline *.c^i & x & y.c^i & \\ \hline v_1^i & T & v_1^i & 1 \\ & \cdots & & \\ v_{n^i}^i & T & v_{n^i}^i & 1 \\ \overline{\emptyset} & F & \overline{\emptyset} & 1 \\ \hline \end{array}, \begin{array}{|c|c|c|c|} \hline *.c^i & x & z.c^i & \\ v_1^i & F & v_1^i & 1 \\ & \cdots & & \\ v_{n^i}^i & F & v_{n^i}^i & 1 \\ \overline{\emptyset} & T & \overline{\emptyset} & 1 \\ \hline \end{array} \right\}$$

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

5. 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`.

$z.c^i$. 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:

$$
T[\texttt{prim_dist}(p_1, \ldots, p_n)] = \begin{array}{|c|c|}\hline * & \\ \hline \end{array}\ \begin{array}{|c|c|}\hline 1 & p_1 \\ \hline \multicolumn{2}{c}{\ldots} \\ \hline n & p_n \\ \hline \end{array}.
$$

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

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

$$
\bigcup_{i=1}^{m} \bigcup_{j=1}^{n} \begin{array}{|c|c|c|c|}\hline *.c^i & z & x_j.c^i & \\ \hline v_1^i & j & v_1^i & 1 \\ \hline \multicolumn{4}{c}{\ldots} \\ \hline v_{n^i}^i & j & v_{n^i}^i & 1 \\ \hline \emptyset & \{j\} & \emptyset & 1 \\ \hline \end{array}.
$$

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

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

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

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 $f.x_i$.[6]

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

6. 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.

factor A_i^j. Then we can write

$$B_i = \bigcup_{j=1}^{m} \begin{array}{|c|c|c|} \hline F & *, f_i.x_1, \ldots, f_i.x_n & \\ \hline f_i & r_1^j & p_1^j \\ & \cdots & \\ f_i & r_{\ell j}^j & p_{\ell j}^j \\ \overline{\{f_i\}} & \overline{\emptyset} \text{ for all} & 1 \\ \hline \end{array}$$

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:

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

14.8 Lessons Learned and Conclusion

The IBAL implementation represents the culmination of several years of investigation, that begin with the original "stochastic Lisp" paper [9] and continued with the SPOOK system [17]. 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 chapter 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 chapter.
- 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. [9] 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 chapter 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 fifty test examples, all of which the inference engine is able to handle. IBAL's tutorial and reference manuals are both over twenty pages long.

Of course, there are many models for which the techniques presented in this chapter 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

[1] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. In *IEEE/ACM International Conference on Computer-Aided Design*, 1993.

[2] T. Dean and K. Kanazawa. A model for reasoning about persistence and causation. *Computational Intelligence*, 5:142–150, 1989.

[3] R. Dechter. Bucket elimination : a unifying framework for probabilistic inference. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*, 1996.

[4] D. Heckerman and J. S. Breese. A new look at causal independence. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*, 1994.

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

[6] D. Koller and A. Pfeffer. Semantics and inference for recursive probability models. In *Proceedings of the National Conference on Artificial Intelligence*, 2000.

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

[8] D. Koller and A. Pfeffer. Probabilistic frame-based systems. In *Proceedings of the National Conference on Artificial Intelligence*, 1998.

[9] D. Koller, D. McAllester, and A. Pfeffer. Effective Bayesian inference for stochastic programs. In *Proceedings of the National Conference on Artificial Intelligence*, 1997.

[10] K. B. Laskey and S. M. Mahoney. Network fragments: Representing knowledge for constructing probabilistic models. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*, 1997.

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

[12] S. Muggleton. Stochastic logic programs. *Journal of Logic Programming*, 2001. Accepted subject to revision.

[13] L. Ngo and P. Haddawy. Answering queries from context-sensitive probabilistic knowledge bases. *Theoretical Computer Science*, 1996.

[14] H. Pasula and S. Russell. Approximate inference for first-order probabilistic languages. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 2001.

[15] A. Pfeffer. Repeated observation models. In *Proceedings of the National Conference on Artificial Intelligence*, 2004.

[16] A. Pfeffer. *Probabilistic Reasoning for Complex Systems*. PhD thesis, Stanford Univeristy, 2000.

[17] A. Pfeffer, D. Koller, B. Milch, and K. T. Takusagawa. SPOOK: A system for probabilistic object-oriented knowledge representation. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*, 1999.

[18] D. Pless and G. Luger. Toward general analysis of recursive probability models. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*, 2001.

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

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

[21] S. Sanghai, P. Domingos, and D. Weld. Dynamic probabilistic relational models. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 2003.

[22] T. Sato and Y. Kameya. Parameter learning of logic programs for symbolic statistical modeling. *Journal of Artificial Intelligence Research*, 15:391–454, 2001.

[23] D. J. Spiegelhalter, A. Thomas, N. Best, and W. R. Gilks. BUGS 0.5 : Bayesian inference using Gibbs sampling manual. Technical report, Institute of Public Health, Cambridge University, 1995.