IBAL: An Expressive, Functional Probabilistic Modeling Language

Avi Pfeffer

Division of Engineering and Applied Sciences
Harvard University

Abstract

There has been much interest in recent years in expressive languages for probabilistic modeling. This paper presents a new approach to developing a powerful, expressive and general probabilistic modeling language. It presents a language, named IBAL, that looks like a stochastic functional programming language. A program in the language is a model of a system; the meaning of the program is the probability distribution over outputs of the program. After presenting the syntax of IBAL, the paper presents examples that demonstrate the expressiveness, compositionality, modularity and ability of the language to represent structure. A declarative semantics is presented that defines the meaning of a program in terms of distributions. A powerful and general inference algorithm is presented that translates a program into factors that can be used in variable elimination. When applied to standard frameworks such as Bayesian networks, hidden Markov models and stochastic context free grammars, IBAL's algorithm reduces to standard algorithms for those frameworks. Thus, when a new framework is modeled in IBAL, an effective inference algorithm is obtained for free.

Key words:
Probabilistic modeling, Expressive languages, Functional languages, Inference

1 Introduction

Generative probabilistic modeling languages, such as Bayesian networks [Pearl, 1988], hidden Markov models [Rabiner, 1989], and stochastic grammars [Charniak, 1993], have achieved great success in artificial intelligence. These frameworks provide structured representation and inference algorithms that allow

Email address: avi@eecs.harvard.edu (Avi Pfeffer).

probabilistic models to be represented naturally and compactly, and reasoned about efficiently. In recent years, there has been much interest in extending the expressive power of probabilistic modeling languages. New languages have been developed (e.g. [Koller and Pfeffer, 1998, Kersting and de Raedt, 2000, Milch et al., 2005, Laskey and Costa, 2005]) that support the representation of the world in terms of objects and relationships, and merge probabilistic representations with traditional logical formalisms.

This paper presents a new approach to developing a powerful, expressive and general probabilistic modeling language. We present a language named IBAL (pronounced "eyeball") for representing generative probabilistic models over discrete variables. IBAL looks like a programming language, but it is not a programming language in the ordinary sense. A program in the language is not a sequence of instructions to be executed, but rather a model of how the world is generated. The meaning of the program is a probability distribution over the generated world. The key idea is that describing a generative process by means of a program allows the full power of programming languages to be harnessed for probabilistic modeling.

IBAL is a functional language: functions are first-class objects in the language, and a key structuring element. There are many advantages to representing models as programs, and particularly functional ones. These include:

- Naturalness: a model in IBAL is an explicit representation of a generative process.
- Expressiveness: IBAL takes a Turing complete programming language and adds stochastic choice. Any generative process that can be described by a program with stochasticity can be represented in IBAL.
- Ability to represent structure: programming languages provide many features that support structured representations. In particular, IBAL can represent conditional independence as used in Bayesian networks using variable bindings and references, functional decomposition as used in stochastic context free grammars using recursive functions, context specific independence and causal independence using conditionals, and object structure from object-oriented Bayesian networks and probabilistic relational models using structured data types and functions.
- Modularity: a part of a model encapsulated within a function can be modified without affecting the rest of the model. Similarly, new functions and data structures can be added to a model at any time.
- Compositionality: individual model components can be combined in well-defined and meaningful ways to create more complex models. For example, one can define a function that takes two entities as inputs that themselves have been defined by functions, and produces a new entity that depends on both of them.
- Combination functions: Functions can be defined that operate on other

structures, including functions, to produce very clear and compact specifications of high level concepts and models. An example of this will be presented in Section 3.3.
- Uniform treatment: all structures, from the most low level to the high level are represented as language constructs. Thus they are all handled in the same way. When new structures are developed, they only need to be coded in terms of the language constructs; no special mechanisms need to be developed.
- A rich type system: tuple types can represent objects, and IBAL supports algebraic data types such as lists and trees. Since functions are first-class objects, functional types can be used in other types. This allows high level types such as a Markov model type to be created. IBAL also provides automatic type inference.

IBAL is a declarative language. An IBAL model is a description of the world. Once the description has been written, reasoning is taken care of by the inference engine. IBAL's inference engine combines techniques from a number of frameworks, together with new techniques of its own. In particular, when the inference engine is applied to a number of existing frameworks, such as Bayesian networks, object-oriented Bayesian networks, hidden Markov models, and stochastic context free grammars, the inference process reduces to standard inference algorithms for these frameworks.

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 is structured as follows. Section 2 formally presents the syntax of the IBAL language. Section 3 contains a number of examples illustrating the use and power of the language. Section 4 follows with a specification of the declarative semantics of the language. Section 5 describes IBAL's inference algorithm. Section 6 contains a discussion of related work, and Section 7 concludes.

2 The IBAL language

In this section, we formally define the core of the IBAL language. In the definitions of the language, we use the following notation:

- a denotes a name, which could be the name of a variable, or the name of a component of a tuple

- c denotes a chain (defined below)
- s denotes a symbolic constant
- x denotes a value (defined below)
- ν denotes an environment (defined below)
- ϵ denotes an expression (defined below)
- π denotes a pattern (defined below)
- p denotes a probability

The following concepts are defined recursively in terms of each other. A *value* is any of

s	Symbol
$\langle a_1 : x_1, ..., a_n : x_n \rangle$	Tuple
$\{\text{formals} : \{a_1, ..., a_n\},\ \text{body} : \epsilon,\ \text{environment} : \nu\}$	Closure

Values other than tuples are called *simple*. Note that functions are first-class objects in the language. Closures, representing functions, can be components of tuples, can be passed as arguments to functions, and can be returned from functions. If $x = \langle a_1 : x_1, ..., a_n : x_n \rangle$, the notation $x.a_i$ denotes x_i. This can be extended to chains: a *chain* is a sequence of names separated by dots. If $c = a_{i_1}...a_{i_{m-1}}.a_{i_m}$ is a chain, $x.c$ denotes the i_m-th component of $x.a_{i_1}...a_{i_{m-1}}$. If c is empty, $x.c$ is equal to x.

An *environment* is a mapping from names to values. The notation $\nu[a]$ denotes the value associated with a in ν. The notation $\nu[a_1/x_1, ..., a_n/x_n]$ denotes the environment formed by extending ν by associating each a_i with x_i.

A *pattern* is any of

s	Constant pattern
$\langle a_1 : \pi_1, ..., a_n : \pi_n \rangle$	Tuple pattern
_	Any (underscore)

The notation $\pi.c$ is defined in a similar manner to values. A value x matches a pattern π, denoted $x \models \pi$, if and only if one of the following cases holds:

- x is any value, and π is _.
- x and π are both equal to a constant s.
- x is $\langle a_1^1 : x_1^1, ..., a_{n_1}^1 : x_{n_1}^1 \rangle$ and π is $\langle a_1^2 : \pi_1^2, ..., a_{n_2}^2 : \pi_{n_2}^2 \rangle$ and $\forall a_i^2 : \pi_i^2, \exists a_j^1 : \pi_j^1$ such that $a_j^1 = a_i^2$ and $x_j^1 \models \pi_i^2$. In this third case, the pattern need not mention all fields in the value. We only require that for each field mentioned by the pattern, there is a field in the value such that the corresponding value matches the corresponding pattern.

Intuitively, an expression defines a computation that produces a value. In IBAL, the value produced is stochastic. Thus an expression represents a *stochastic experiment*. Most of the expression types will be familiar from ordinary programming languages. The two new ones are `dist` expressions, which define stochastic choice, and `obs` expressions, which allow conditioning on observations. The different types of IBAL expressions are as follows:

$'s$	Symbolic constant
$a.c$	Variable
$\langle a_1 = \epsilon_1, ..., a_n = \epsilon_n \rangle$	Tuple construction
`if` ϵ_1 `then` ϵ_2 `else` ϵ_3	Conditional
`dist` $[p_1 : \epsilon_1, ..., p_n : \epsilon_n]$	Stochastic choice
`let` $a = \epsilon_1$ `in` ϵ_2	Variable binding
`fix` $a_0(a_1, ..., a_n) = \epsilon$	Function definition
$\epsilon_0(\epsilon_1, ..., \epsilon_n)$	Function application
$\epsilon_1 == \epsilon_2$	Equality test
`obs` π `in` ϵ	Observation

The symbolic constant expression $'s$ represents the experiment that always evaluates to s.

The variable expression $a.c$ evaluates to $x.c$, where x is the value of a in the environment.

A tuple construction expression $\langle a_1 = \epsilon_1, ..., a_n = \epsilon_n \rangle$ creates a tuple in which each component a_i has the value of the corresponding subexpression ϵ_i.

A conditional expression `if` ϵ_1 `then` ϵ_2 `else` ϵ_3 represents a stochastic experiment in which ϵ_1 is evaluated first. If ϵ_1 evaluates to true, then the value of ϵ_2 is produced, otherwise ϵ_3 is produced.

An expression `dist` $[p_1 : \epsilon_1, ..., p_n : \epsilon_n]$ represents stochastic choice. The value of one of the ϵ_i is chosen, each with probability p_i.

In a variable binding expression `let` $a = \epsilon_1$ `in` ϵ_2, the environment is extended by binding a to the value of ϵ_1, and the value of ϵ_2 in the extended environment is produced.

A function definition expression `fix` $a_0(a_1, ..., a_n) = \epsilon$ produces a function named a_0, with the given formal arguments and body. The function name may appear recursively in the body of the function.

In the expression $\epsilon_0(\epsilon_1, ..., \epsilon_n)$, the functional value of ϵ_0 is applied to the values of $\epsilon_1, ..., \epsilon_n$. Note that ϵ_0 may be stochastic, so that there is more than one possible function to apply.

An equality test $\epsilon_1 == \epsilon_2$ produces `true` if the values of ϵ_1 and ϵ_2 are equal.

An observation `obs` π `in` ϵ produces the value of ϵ, conditioned on it matching π. This can be understood most simply as defining a rejection process within the stochastic experiment. The value of ϵ is produced repeatedly until a value that matches π is produced.

Precise semantics for all these expression types will be given in Section 4.

It is important to note that in an expression of the form `let` a `=` ϵ_1 `in` ϵ_2 the variable a is assigned a specific value in the experiment; any stochastic choices made while evaluating ϵ_1 are resolved, and the result is assigned to a. For example, consider

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

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 `if z then z else false`. 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.

It is important to emphasize that while a program can be readily understood as defining a stochastic process for computing the value of an expression, the meaning of the program is not the value produced, but the *probability distribution* over the value. When we perform inference on the program, we will not be generating a single value but a probability distribution over values. Thus IBAL is a language for representing a probabilistic model by functionally describing its generative process.

In addition to the core language described above, IBAL provides a good deal of syntactic sugar to make it easier to represent models. The added features include easier syntax for function definitions, Boolean and integer values and operations, error expressions, comments, and support for algebraic data types such as lists and trees. These extended features can usually be implemented in terms of the core language. To keep things simple we will only define the semantics and describe the inference algorithm for the core language. However we will use the extended features freely in the examples, since they make them clearer and more readable. For the most part, their meaning will be clear from context, or will be explained where they occur.

One feature we will describe here is block notation. In the description above, a program is an expression, which is composed of subexpressions. A real program contains a series of definitions and observations in a block. Each definition consists of a variable name, and an expression defining its value. Optionally, the definition may define a function, in which case its name, formal arguments, and body will be provided. Block notation may also include observations about the variables defined in the block.

Another extended language construct worth pointing out here is the `case` expression. The general syntax of `case` expressions is

$$\texttt{case}\ \epsilon_0\ \texttt{of}$$
$$\#\pi_1 : \epsilon_1$$
$$\ldots$$
$$\#\pi_n : \epsilon_n$$

where the π_i are extended patterns and the ϵ_i are expressions. An extended pattern is like an ordinary pattern described earlier, except that it can also contain a variable name. A variable name is similar to an "any" pattern, in that all values match it, except that whan a value is matched to a variable pattern, the variable is bound to the value, for use inside the expression ϵ_i. The meaning of a `case` expression, in terms of a stochastic experiment, is to begin by evaluating ϵ_0, to produce x_0. Then x_0 is matched to each of the patterns in turn. If x_0 matches π_1, the result of the experiment is the result of ϵ_1, where the environment is extended with any bindings produced while matching x_1 to π_1. If x_0 does not match π_1, the process continues with π_2, and so one. It is an error for the value not to match any pattern.

In this paper, we assume that all programs are well typed, that all variables are defined before they are used, that the same name does not appear more than once in a tuple, and that all chains refer to actual components that exist. IBAL is a strongly typed language and the system performs automatic type inference.

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` 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
```

`flip` p is shorthand for a stochastic choice expression that with probability p produces `true`, and with remaining probability produces `false`.

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
  | burglary & 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.

```
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 and 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.

PRMs 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`, `student1`, `student2` and `perf1` through `perf4`) 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. Similarly, the `performance` function relates a student to a course.

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

course(p, f) = {
  well_taught = (p.clear | ~ f.hard) & flip 0.9;
  teacher = p;
};

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

performance(s, c) = {
  understands =
    if s.hard_working
    then case <s.smart, c.well_taught> of
    # <true, true> : flip 0.95
```

```
    # <false, false> : flip 0.35
    # _ : flip 0.7
    else case s.smart of
    # true : flip 0.7
    # false : flip 0.1;

  exam_grade =
    case <understands, s.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, s.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()
student2 = student()

perf1 = performance(student1, course1)
perf2 = performance(student1, course2)
perf3 = performance(student2, course1)
perf4 = performance(student2, course2)

observe perf1.homework_grade = 'A
observe perf2.exam_grade = 'C
observe perf3.homework_grade = 'B
```

∎

Example 3.3:

This example illustrates the power of higher-order functions. A transition model in a dynamic system can be captured in IBAL by a function from states to states. Suppose we wish to create a library of transition models, including ones that create more complex models out of simpler models. We can write a higher-order function that takes simpler models as its input and combines them in some way to produce more complex models. For example, we might try to write a `convoy` function that will combine transition models for individual vehicles, to produce a transition model for a convoy of vehicles.

The inputs to the `convoy` function will be of two kinds. The first will apply to a vehicle that is at the lead of a convoy, whose new position is determined only by its own previous position. We will create such a model for each vehicle:

```
lead_car (prev) = ...

lead_truck (prev) = ...
```

For a vehicle that is following another vehicle, its new position will be determined not only by its own previous position but also by the position of the vehicle in front of it. We create such a model for each type of vehicle:

```
car (prev, ahead) = ...

truck (prev, ahead) = ...
```

Now, we have to specify how to combine these models to produce a convoy model. The `convoy` function will take two inputs: a lead vehicle, and a list of subsequent vehicles (all of which are functions). It produces a transition function that takes a list of states of all vehicles and produces a list of states of all vehicles. The function is defined as follows: [1]

```
convoy(leadVehicle, restVehicles) =
  fun states ->
    let process(ahead, states, vehicles) =
       case <states, vehicles> of
       # < [], [] > -> []
       # < firstState :: restStates, firstVehicle :: restVehicles > ->
               let pos = firstVehicle(firstState, ahead) in
               pos :: process(pos, restStates, restVehicles)
       # _ -> error "length mismatch"
    in
    let firstPos = leadVehicle(head(states)) in
    firstPos :: process(firstPos, tail(states), restVehicles)
```

[1] `fun` a `->` ϵ defines the function whose formal argument is a and whose body is ϵ.

In this example we use standard syntax for lists: `[]` denotes the empty list, x`::`ℓ denotes the list in which x is consed onto ℓ, and `[`x_1`,...,`x_n`]` denotes the list consisting of $x_1, ..., x_n$. We assume that `head` and `tail` have been defined.

The heart of the `convoy` function is a recursive `process` function that takes the position of a vehicle ahead of the vehicles to be processed, a list of states of vehicles to be processed, and the vehicles themselves. In the base case, both lists are empty, and the result is the empty list of states. A case in which one list is empty produces an error; we assume that the length of the `states` list equals the number of vehicles. In the remaining case, we apply the first vehicle function to the first state, and the position of the vehicle ahead, to get a new position for the first vehicle. We then call `process` recursively, using this position as the position ahead of the remaining sequence of vehicles. In the result, we just tack on the new position obtained to the result of the recursive call to `process`. Once `process` has been defined, `convoy` can be defined easily. We simply apply the lead vehicle function to the first state to get the first new position, and tack it on to the result of calling `process` on the remaining vehicles, using the first new position as `ahead` in `process`.

Once `convoy` has been written, creating instances of convoy models for particular sequences of vehicles is tremendously easy. For example, you only have to write

```
convoy(leadCar, [ car, truck, truck, truck ])
```

for one convoy, and

```
convoy(leadTruck, [ car, car, truck ])
```

for another. There is a nice division of labor here. The `convoy` function needs to be written by someone who understands functional programming, but it is relatively easy to write, and the programmer does not need to know about how individual vehicles are modeled. Furthermore, this is a one-time cost. The individual vehicle functions need to be written by someone who knows about their transition models, but the functions themselves are straightforward and the modeler does not need to be a sophisticated programmer or know about higher order functions. In addition, new types of vehicles can easily be added modularly by writing their leading and following functions. Finally, once the vehicle functions have been developed, anyone, even a non-programmer, can use them to instantiate new convoy models on the fly.

∎

Example 3.4:

The next example illustrates IBAL's power to model first-order structures and

relationships. It concerns a stochastic blocks world. A robot may perform three actions: `try_pickup`, which tries to pick a block up off the table, `try_puton`, which tries to put a block the robot is holding on top of another block, and `try_drop`, which tries to drop the block the robot is holding on the table. The effects of the actions are stochastic. Blocks are represented by integers from 0 to n-1. The state of the world has two components: `on`, which is a list of length n in which component i is the block on which block i is sitting (-1, if block i is on the table, and -2 if it is currently being held by the robot); and `holding`, which is the current block being held (-1 if no block is being held).

The code begins by definining two useful list utilities: `set`, which changes the value of one element of a list, and `for_all`, which checks if a property holds for every element of a list.

```
set (n, x, xs) =
  case xs of
  # [] : error "range error"
  # y :: ys :
      if n == 0
      then x :: ys
      else y :: set (n - 1, x, ys)

for_all (f, xs) =
  case xs of
  # [] : true
  # y :: ys : f (y) & for_all (f, ys)
```

Whether or not a block is clear is not stored as part of the state. Instead it is derived from the state using the following function:

```
clear (block, state) =
  ~ (state.holding == block) &
  for_all (fun x -> ~ (x == block), state.on)
```

We can now define the different operations. For each operation, we will define a successful version and one or more failed versions. We will then define the function for trying the operation. If the preconditions of the operation are not met, the state stays the same, otherwise it is a stochastic choice between the successful and failed versions. [2]

```
successful_pickup (block, state) =
  { on = set (block, -2, state.on); holding = block; }
```

[2] A failed `pickup` drops the block being picked up on the table. `puton` can fail in two ways: it can miss, in which case the block being held is dropped on the table, or it can completely fail, in which case nothing changes.

```
failed_pickup (block, state) =
  { on = set (block, -1, state.on); holding = -1; }

try_pickup (block, state) =
  if ~(state.holding == -1) | ~(clear (block, state))
  then state
  else dist [ 0.1 : failed_pickup (block, state),
              0.9 : successful_pickup (block, state) ]

successful_puton (target_block, state) =
  { on = set (state.holding, target_block, state.on);
    holding = -1; }

missed_puton (state) =
  { on = set (state.holding, -1, state.on);
    holding = -1; }

failed_puton (state) = state

try_puton (target_block, state) =
  if (state.holding == -1) | ~(clear (target_block, state))
  then state
  else dist [ 0.09 : missed_puton(state),
              0.06 : failed_puton(state),
              0.85 : successful_puton(target_block, state) ]

successful_drop (state) =
  { on = set (state.holding, -1, state.on);
    holding = -1; }

failed_drop (state) = state

try_drop (state) =
  if state.holding == -1
  then state
  else dist [ 0.08 : failed_drop(state),
              0.92 : successful_drop(state) ]
```

4 Semantics

An intuitive semantics for IBAL can be obtained from the idea that an IBAL program defines a sampling process, that will either fail because an observation is violated, or else succeed and produce a value. The meaning of the program is then the distribution over the value produced. It is easy to define a sampling process to capture this. However, it is more satisfying to provide a declarative semantics, in which the meaning of a program is directly specified in terms of a probability distribution, without resorting to the idea of executing a process. In this section we provide such a semantics. Nevertheless, keeping the sampling process in mind will help ground our intuitions when defining the distributional semantics.

Note that the semantics has changed from previous incarnations of IBAL. In the old semantics, the meaning of an expression was defined to be the probability distribution over its output given the observations inside the expression. In the new semantics, the meaning of an expression is defined by two functions. Both functions take an expression ϵ and an environment ν as arguments. An *environment* is simply a function from chains to values. The first function, denoted $P_e[\epsilon, \nu]$, represents the probability of the observations within the expression, given that the variables in the environment take on their given values. The second function, denoted $P[\epsilon, \nu]$, denotes the probability distribution over values produced by the expression, conditioned on the observations within the expression holding. The notation $P[\epsilon, \nu](x)$ will denote the probability of x under this distribution. Similarly, $P[\epsilon, \nu](\pi)$ will denote the probability that a value drawn from this distribution matches pattern π. These two components are defined recursively in terms of each other. Separating things out into these two components results in a much cleaner and clearer semantics.

The semantics is defined by equations. In the base cases, such as a constant or variable expression, the functions are defined directly. For expressions that are defined in terms of other expressions, the meaning of the expression is produced from the meanings of its subexpressions, which are combined in a particular way. The semantics is defined by the following equations. In the definition of $P[\epsilon, \nu]$, only those values with positive probability will be mentioned.

A constant expression has no observations, so trivially the probability of the observations is 1. The expression produces the constant itself with probability 1.

$$P_e['s, \nu] = 1$$
$$P['s, \nu](s) = 1$$

In a variable expression the observations have probability 1 of being satisfied.

(We assume the program is well typed, so the variable always exists in the environment.) The distribution defined by the expression assigns probability 1 to the value of the variable in the environment.

$$P_e[a, \nu] = 1$$
$$P[a.c, \nu](\nu[a].c) = 1$$

For a tuple expression, the different components are produced independently *given the environment*. They may be dependent on each other, but only if they share variables present in the environment.

$$P_e[\langle a_1 = \epsilon_1, ..., a_n = \epsilon_n \rangle, \nu] = \prod_{i=1}^{n} P_e[\epsilon_i, \nu]$$
$$P[\langle a_1 = \epsilon_1, ..., a_n = \epsilon_n \rangle, \nu](\langle a_1 = x_1, ..., a_n = x_n \rangle) = \prod_{i=1}^{n} P[\epsilon_i, \nu] x_i$$

For `if` expressions we have

$$P_e[\texttt{if } \epsilon_1 \texttt{ then } \epsilon_2 \texttt{ else } \epsilon_3, \nu] =$$
$$P_e[\epsilon_1, \nu] \{P[\epsilon_1, \nu](\texttt{true}) P_e[\epsilon_2, \nu] + P[\epsilon_1, \nu](\texttt{false}) P_e[\epsilon_3, \nu]\}$$

The intuition behind this is that we want to know the probability that all the observations encountered during evaluation of the program are satisfied. For that to happen, first of all observations in the test must be satisfied, so we have a $P_e[\epsilon_1, \nu]$ term. Then, if the value of the test is `true`, ϵ_2 is evaluated and all observations encountered in it must be satisfied. This happens with probability $P[\epsilon_1, \nu](\texttt{true})$, and when it happens we have a $P_e[\epsilon_2, \nu]$ term. Similar considerations apply to the case where the ϵ_3 is evaluated. We also have

$$P[\texttt{if } \epsilon_1 \texttt{ then } \epsilon_2 \texttt{ else } \epsilon_3, \nu](x) =$$
$$\tfrac{1}{Z} \{P[\epsilon_1, \nu](\texttt{true}) P_e[\epsilon_2, \nu] P[\epsilon_2, \nu](x) + P[\epsilon_1, \nu](\texttt{false}) P_e[\epsilon_3, \nu] P[\epsilon_3, \nu](x)\}$$

where Z is a normalizing factor. The reasoning behind this is that $P_e[\epsilon_2, \nu] P[\epsilon_2, \nu](x)$ is the joint probability that the evidence in ϵ_2 is satisfied, and evaluating ϵ_2 in ν produces x. Thus $P[\epsilon_1, \nu](\texttt{true}) P_e[\epsilon_2, \nu] P[\epsilon_2, \nu](x)$ is the probability, given that the evidence in ϵ_1 is satisfied, that ϵ_1 evaluates to `true`, ϵ_2 evaluates to x, and the evidence in ϵ_2 is satisfied. Therefore the term in curly braces is the total probability that the `if` expression evaluates to x and the evidence in whichever part is evaluated is satisfied, given that the evidence in ϵ_1 is

satisfied. Note that the normalizing factor Z is equal to

$$\begin{aligned}
&\textstyle\sum_x \{P[\epsilon_1,\nu](\texttt{true})P_e[\epsilon_2,\nu]P[\epsilon_2,\nu](x) + P[\epsilon_1,\nu](\texttt{false})P_e[\epsilon_3,\nu]P[\epsilon_3,\nu](x)\} \\
&= P[\epsilon_1,\nu](\texttt{true})P_e[\epsilon_2,\nu] + P[\epsilon_1,\nu](\texttt{false})P_e[\epsilon_3,\nu] \\
&= \tfrac{P_e[\texttt{if } \epsilon_1 \texttt{ then } \epsilon_2 \texttt{ else } \epsilon_3,\nu]}{P_e[\epsilon_1,\nu]}
\end{aligned}$$

Similar reasoning applies to many of the other expression forms. For example, for `dist` expressions we have

$$\begin{aligned}
&P_e[\texttt{dist } [p_1 : \epsilon_1, ..., p_n : \epsilon_n], \nu] = \textstyle\sum_{i=1}^n p_i P_e[\epsilon_i, \nu] \\
&P[\texttt{dist } [p_1 : \epsilon_1, ..., p_n : \epsilon_n], \nu](x) = \tfrac{1}{Z} \textstyle\sum_{i=1}^n p_i P_e[\epsilon_i, \nu] P[\epsilon_i, \nu](x)
\end{aligned}$$

where $Z = P_e[\texttt{dist } [p_1 : \epsilon_1, ..., p_n : \epsilon_n], \nu]$.

For a `let` expression $\texttt{let } a = \epsilon_1 \texttt{ in } \epsilon_2$, for the probability of observations we need the probability that the observations in ϵ_1 are satisfied, and then the probability that the observations in ϵ_2 are satisfied in the environment produced by extending the original environment by binding a to whatever value ϵ_1 evaluated to. Therefore we need to sum over all possible values x_1 that ϵ_1 could evaluate to. Similar considerations apply to specifying the distribution defined by the expression.

$$\begin{aligned}
&P_e[\texttt{let } a = \epsilon_1 \texttt{ in } \epsilon_2, \nu] = P_e[\epsilon_1, \nu] \textstyle\sum_{x_1} P[\epsilon_1, \nu](x_1) P_e[\epsilon_2, \nu[a/x_1]] \\
&P[\texttt{let } a = \epsilon_1 \texttt{ in } \epsilon_2, \nu](x) = \tfrac{1}{Z} \textstyle\sum_{x_1} P[\epsilon_1, \nu](x_1) P_e[\epsilon_2, \nu[a/x_1]] P[\epsilon_2, \nu[a/x_1]](x)
\end{aligned}$$

where $Z = \frac{P_e[\texttt{let } a=\epsilon_1 \texttt{ in } \epsilon_2,\nu]}{P_e[\epsilon_1,\nu]}$.

When we define a function, the meaning of the defined function is a closure, specifying formal arguments, a function body, and an environment in which to evaluate the function. This environment is the current environment extended by binding the function name to the closure itself.

$$\begin{aligned}
&P_e[\texttt{fix } a_0(a_1, ..., a_n) = \epsilon, \nu] = 1 \\
&P[\texttt{fix } a_0(a_1, ..., a_n) = \epsilon, \nu](x) = 1 \\
&\quad \text{where } x = \{\text{formals: } \{a_1, ..., a_n\}, \text{body: } \epsilon, \text{env: } \nu[a_0/x]\}
\end{aligned}$$

Recall that in a function application, the function to be applied may itself be stochastic. We therefore have to sum over all its possible values. We also have to sum over all possible values of the function arguments. Once those have to

be specified, we examine the body of the function being applied, in the new environment produced by extending the closure environment by binding the arguments to their specified values.

$$
\begin{aligned}
&P_e[\epsilon_0(\epsilon_1, ..., \epsilon_n), \nu] = \\
&\quad P_e[\epsilon_0, \nu] \textstyle\prod_{i=1}^{n} P_e[\epsilon_i, \nu] \sum_{x_0} P[\epsilon_0, \nu](x_0) \sum_{x_1} ... \sum_{x_n} \prod_{i=1}^{n} P[\epsilon_i, \nu](x_i) P_e[x_0.\text{body}, \nu'] \\
&\quad \text{where } \nu' = x_0.\text{env}[x_0.\text{formals}_1/x_1, ..., x_0.\text{formals}_n/x_n] \\
&P[\epsilon_0(\epsilon_1, ..., \epsilon_n), \nu](x) = \\
&\quad \tfrac{1}{Z} \sum_{x_0} P[\epsilon_0, \nu](x_0) \sum_{x_1} ... \sum_{x_n} P[\epsilon_i, \nu](x_i) P_e[x_0.\text{body}, \nu'] P[x_0.\text{body}, \nu'](x)
\end{aligned}
$$

where ν' is as above, and $Z = \frac{P_e[\epsilon_0(\epsilon_1,...,\epsilon_n),\nu]}{\prod_{i=0}^{n} P_e[\epsilon_i,\nu]}$.

The semantics of equality tests is straightforward, given the above.

$$
\begin{aligned}
&P_e[\epsilon_1 == \epsilon_2, \nu] = P_e[\epsilon_1, \nu] P_e[\epsilon_2, \nu] \\
&P[\epsilon_1 == \epsilon_2, \nu](\texttt{true}) = \textstyle\sum_x P[\epsilon_1, \nu](x) P[\epsilon_2, \nu](x) \\
&P[\epsilon_1 == \epsilon_2, \nu](\texttt{false}) = 1 - P[\epsilon_1 == \epsilon_2, \nu](\texttt{true})
\end{aligned}
$$

Finally, we have the meaning of an observation expression `obs` π `in` ϵ. The probability of observations in the entire expression is the probability of the observation in the embedded expression, times the probability that the value produced matches the given pattern π. The distribution defined by the entire expression is the distribution defined by the embedded expression, conditioned on the value satisfying π. If the probability that the value satisfies π is zero, the meaning is not well defined.

$$
\begin{aligned}
&P_e[\texttt{obs } \pi \texttt{ in } \epsilon, \nu] = P_e[\epsilon, \nu] P[\epsilon, \nu](\pi) \\
&P[\texttt{obs } \pi \texttt{ in } \epsilon, \nu](x) = \begin{cases} \frac{1}{P[\epsilon,\nu](\pi)} P[\epsilon, \nu](x) & \text{if } x \models \pi \\ 0 & \text{otherwise} \end{cases}
\end{aligned}
$$

The semantics as presented here assumes that the program terminates under all possible stochastic choices. We can extend the semantics to cover situations in which a program terminates with probability one, but there exist stochastic choices under which it does not terminate. For example, consider a stochastic grammar that produces arbitrarily long strings, but produces a finite string with probability one. To capture these models, we can define a fixpoint semantics. On level 0, $P_e^0[\epsilon, \nu]$ would be 1, and $P^0[\epsilon, \nu]$ would assign probability zero to every value. Recursively, we would define P_e^i and P^i using the above

equations, with P_e^{i-1} and P^{i-1} appearing on the right hand side. The meaning of the program would be the limit as $n \to \infty$ of P_e^n and P^n.

5 Inference

The goal of probabilistic inference in IBAL is to compute the joint probability of the evidence within a program and the output of the program. That is, given a program ϵ, the goal is to compute $P_e[\epsilon, []]P[\epsilon, []]$, where [] is the empty environment. The reasons this is computed rather than simply $P[\epsilon, []]$ are that it makes for a cleaner inference algorithm with no normalization required, and that sometimes one might be interested to know the probability of observations.

At its heart, IBAL uses *variable elimination* (VE) [Zhang and Poole, 1994, Dechter, 1999]. The core of the algorithm is a set of functions that can transform expressions into factors used in VE. We will first review VE, then describe the particular representation of factors used, and then present the transformation functions for converting programs into factors.

5.1 Variable Elimination

VE is a widely used algorithm using in Bayesian network inference, constraint satisfaction, and other applications. VE works with a sum-of-products expression of the form $\sum_{c_1} \sum_{c_2} ... \sum_{c_n} \prod_{i=1}^{m} F_i$. Each F_i is a factor from some set of variables to a domain. In probabilistic inference a factor is a function from the values of a set of variables to real numbers. We say that the factor *mentions* this set of variables. A factor may mention variables in $c_1, ..., c_n$ and other variables. $c_1, ..., c_n$ are the variables to be eliminated. The goal is to manipulate the sum-of-products expression until all of $c_1, ..., c_n$ have been eliminated, and we are left with a factor that mentions only the remaining variables. The algorithm works as follows:

$\mathbf{F}$ = the set of factors.
$c_1, ..., c_n$ = the variables to eliminate, in some order.
For $i = 1$ to n:
  Let $\mathbf{G} = \{G_1, ..., G_k\}$ be the elements of $\mathbf{F}$ that mention c_i.
  Compute $H = \sum_{c_i} \prod_{j=1}^{k} G_k$.
  $\mathbf{F} \leftarrow (\mathbf{F} - \mathbf{G}) \cup \{H\}$.
Let $\{F_1, ..., F_\ell\}$ be the factors remaining in $\mathbf{F}$.
Return $\prod_{j=1}^{\ell} F_j$.

The choice of elimination ordering is important and can greatly influence the cost of inference. In our implementation we use the commonly used minfill heuristic [Kjaerulff, 1990], but any method for computing elimination orderings can be used, including heuristics such as maximum cardinality [Tarjan and Yannakakis, 1984], exact methods [Shoikhet and Geiger, 1997] and approximate methods [Becker and Geiger, 2001]. However, since we use a different representation of factors to the standard one as described below, it may be that another method is more appropriate for our purposes. Developing algorithms for finding good elimination orderings using our representation of factors is a subject for future research.

The key operations in the algorithm are multiplying two factors, and summing over a variable in a factor. Let F_1 be a factor mentioning $\mathbf{c}_1$ and F_2 be a factor mentioning $\mathbf{c}_2$, where $\mathbf{c}_1$ and $\mathbf{c}_2$ may share variables. Then the product $F_1 \cdot F_2$ is a factor F such that $F(\mathbf{c}_1 \cup \mathbf{c}_2) = F_1(\mathbf{c}_1) \cdot F_2(\mathbf{c}_2)$. For summation, let F be a factor mentioning c_i and other variables $\mathbf{d}$. Then $\sum_{c_i} F$ is a factor G such that $G(\mathbf{d}) = \sum_{c_i} F(c_i, \mathbf{d})$.

5.2 Representation of Factors

The standard representation used for factors in BN inference is a table. A table explicitly enumerates every combination of values of the variables, and assigns to each a real number. This is unsuitable for IBAL for two reasons. First, factors in IBAL exhibit a great deal of context-specific independence [Boutilier et al., 1996]. In context-specific independence, whether or not a variable affects the value of the function may depend on the values of context variables. In some contexts we care about the value of the variable, but in others we don't. A table representation requires to enumerate all cases explicitly, even when many of them are the same. Second, IBAL factors exhibit great sparsity. Many, if not most, of the combinations map to zero. In a table representation, all combinations must be listed. Poole and Zhang [2003] have developed a VE-based inference algorithm for networks with context-specific independence. Here we present a representation that is geared specifically to the structure of IBAL programs. We view the representation of factors as a minor contribution of this paper. While it is different from previous approaches, we do not show that it is better than the approach of Poole and Zhang. However, it is important to present this representation as it will be required to understand the functions that transform expressions into factors, which are the heart of the algorithm.

The IBAL representation of factors is defined as follows:

Definition 5.1: A *domain* is a finite set of values $\{x_1, ..., x_n\}$.

A *test* is either $[\in S]$ or $[\notin \{S\}]$, where S is a domain.

A *key* is a set of chains $\{c_1, ..., c_m\}$.

A *constraint* for a key is a function mapping each chain in the key to a test.

A *row* for a key is a pair (ϕ, p) where ϕ is a constraint for the key, and p is a real number.

A *factor* is a triple $(\mathbf{c}, \mathbf{S}, \mathbf{r})$, where $\mathbf{c}$ is a key, $\mathbf{S}$ is a function mapping chains in $\mathbf{c}$ to domains, and $\mathbf{r}$ is a set of rows for $\mathbf{c}$.

∎

Note that the rows in the factor are not required to be mutually exclusive or exhaustive. The reason for the lack of exhaustivity is sparseness. We don't want the factor to have to include rows that map to zero. The reason for the lack of mutual exclusivity will be explained shortly. A factor represents a function on the chains in the key as follows.

Definition 5.2: Let $F = (\mathbf{c}, \mathbf{S}, \mathbf{r})$ be a factor. Let $\mathbf{x}$ be an assignment of values to the chains in $\mathbf{c}$, i.e. a function that maps each chain to a value.. We say that $\mathbf{x}$ *satisfies* a constraint ϕ if for each chain $c \in \mathbf{c}$, either (1) $\phi(c) = [\in T]$ and $\mathbf{x}(c) \in T$, or (2) $\phi(c) = [\notin T]$ and $\mathbf{x}(c) \in \mathbf{S}(c) - T$. We view ϕ as defining a set and write $\mathbf{x} \in \phi$.

The factor F then defines the following function:

$$F(\mathbf{x}) = \sum_{\rho=(\phi,p)\in\mathbf{r}:\mathbf{x}\in\phi} p.$$

∎

We need to define product and summation operations for factors. We first need the intersection of tests and constraints.

Definition 5.3: The intersection of two tests is defined as

$$[\in S_1] \cap [\in S_2] = [\in S_1 \cap S_2]$$
$$[\in S_1] \cap [\notin S_2] = [\in S_1 - S_2]$$
$$[\notin S_1] \cap [\in S_2] = [\in S_2 - S_1]$$
$$[\notin S_1] \cap [\notin S_2] = [\notin S_1 \cup S_2]$$

Let $\mathbf{c}_1$ and $\mathbf{c}_2$ be two keys, and let $\mathbf{c}$ be $\mathbf{c}_1 \cup \mathbf{c}_2$. If ϕ_1 is a constraint for $\mathbf{c}_1$, the *extension* of ϕ_1 to $\mathbf{c}$, denoted $E(\phi_1)$, is a constraint for $\mathbf{c}$ in which $E(\phi_1)(c) = \phi_1(c)$ if $c \in \mathbf{c}_1$, and $E(\phi_1)(c) = [\notin \emptyset]$ if $c \notin \mathbf{c}_1$. We similarly define $E(\phi_2)$ for a constraint ϕ_2 on $\mathbf{c}_2$. We then define $\phi_1 \cap \phi_2$ to be the constraint ϕ on $\mathbf{c}$ in which $\phi(c) = E(\phi_1)(c) \cap E(\phi_2)(c)$. ∎

Using this, the product of two factors can now be defined.

Definition 5.4: Let $F_1 = (\mathbf{c}_1, \mathbf{S}_1, \mathbf{r}_1)$ and $F_2 = (\mathbf{c}_2, \mathbf{S}_2, \mathbf{r}_2)$ be two factors. The product $F_1 \cdot F_2$ is a factor $F = (\mathbf{c}, \mathbf{S}, \mathbf{r})$, defined as follows:

$$\begin{aligned}
\mathbf{c} &= \mathbf{c}_1 \cup \mathbf{c}_2 \\
\mathbf{S}(c) &= \begin{cases} \mathbf{S}_1(c) & \text{if } c \in \mathbf{c}_1 - \mathbf{c}_2 \\ \mathbf{S}_2(c) & \text{if } c \in \mathbf{c}_2 - \mathbf{c}_1 \\ \mathbf{S}_1(c) \cap \mathbf{S}_2(c) & \text{if } c \in \mathbf{c}_1 \cap \mathbf{c}_2 \end{cases} \\
\mathbf{r} &= \{(\phi_1 \cap \phi_2, p_1 * p_2) : (\phi_1, p_1) \in \mathbf{r}_1, (\phi_2, p_2) \in \mathbf{r}_2, \phi_1 \cap \phi_2 \neq \emptyset\}
\end{aligned}$$

∎

In words, the rows of the product are formed from taking the cross product of the rows in the arguments. For each pair of rows, we multiply the associated probabilities. However we throw out all pairs of rows whose constraints are inconsistent. This helps keep the resulting factor reasonably small. In fact, we have observed that in some cases the product has fewer rows than one of the arguments. When applied to factors in which each combination of values is enumerated as in a table, this reduces to the standard product operation for the table representation.

Summation is less straightforward. Previously we used a summation operation which preserved mutual exclusivity. Implementing this operation required taking differences between constraints. Unfortunately the difference between two constraints cannot necessarily be expressed as a single constraint but must be a list of constraints. Furthermore, this operation was quadratic in the number of rows in the factor, and was a major bottleneck. Therefore we have abandoned this approach in favor of a linear time operation that does not preserve mutual exclusivity. Even though this operation tends to produce larger factors, we have found inference using it to be much more efficient. Our summation operation is defined as follows.

Definition 5.5: Let $F_1 = (\mathbf{c}_1, \mathbf{S}_1, \mathbf{r}_1)$ be a factor, and let c be the chain that we are summing over. $\sum_c F_1$ is the factor $F = (\mathbf{c}, \mathbf{S}, \mathbf{r})$ defined as follows: $\mathbf{c}$ is $\mathbf{c}_1 - \{c\}$. $\mathbf{S}$ is the restriction of $\mathbf{S}_1$ to $\mathbf{c}$. Let $R(\mathbf{r}_1)$ be

$$\{(\phi, p) : (\phi_1, p) \in \mathbf{r}_1 \text{ and } \phi \text{ is the restriction of } \phi_1 \text{ to } \mathbf{c}\}.$$

Then $\mathbf{r}$ is

$$\{(\phi, \sum_{\rho=(\phi,p)\in R(\mathbf{r}_1)} p) : \phi \text{ appears in some row in } R(\mathbf{r}_1)\}$$

∎

More operationally, in order to sum c out of F_1, we first delete c from the key, delete its domain from the domain function, and for each row in F_1, delete its test from the row constraint. We then take all the constraints that appear in the resulting rows, ignoring what they used to say about c. For each such constraint, we create a row in the new factor that associates the constraint with the sum of the numbers in all the rows containing that constraint. When applied to a factor in which all possible combinations of values are enumerated as in a tabular representation, this exactly reduces to the ordinary summation operation.

For illustrative purposes, we will present factors using a table. The first row of the table will be the key. The second row will show the domain for each chain. The table will contain a row for each row in the factor. The entry in the column belonging to a chain will be the test for that chain. On the right hand side will appear the probability associated with the row.

5.3 Factor-Producing Functions

The main task of the inference algorithm is to take an expression and transform it into a set of factors, together with a set of temporary variables, such that performing variable elimination on the factors, eliminating the temporary variables, produces the required answer.

The inference algorithm reasons about the possible values a variable or an expression can take. A *support* of a variable is a domain of values that includes all values that the variable can take with positive probability. (It may contain other values in some cases, but in general we would like the support to be as small as possible.) The support of an expression is defined similarly. A *support environment* maps chains to supports. An *instance* of a support environment σ is an environment ν such that for each chain c associated with a support in σ, $\nu(c) \in \sigma(c)$. The support is used to streamline the computation. Instead of having to consider all conceivable values for a variable, we only have to consider those that actually have positive probability. The value of this will be illustrated below.

We introduce the following notation for different kinds of chains. The notation $\star$ denotes a variable representing the outcome of the expression. Any chain beginning with $\star$ represents a component of the value of the expression. $\star\star$ will denote the set of all chains beginning with $\star$ mentioned in a set of factors. $a.c$ (where c is possibly empty) will denote a chain beginning with program variable a, and $\mathbf{a}$ will denote the set of all chains beginning with a program variable in a set of factors. z will denote a temporary variable introduced by the algorithm that does not appear in the program, and $\mathbf{z}$ will denote the set of

all chains in a set of factors beginning with temporary variables. Whenever we introduce a temporary variable, we introduce a fresh variable that is distinct from all other variables, to avoid conflicts.

If $\mathbf{F}$ is a set of factors mentioning temporary chains $\mathbf{z}$, we let $Q_{\mathbf{F}}(\star\star|\mathbf{a}) = \sum_{\mathbf{z}} \prod \mathbf{F}$. This denotes the joint probability distribution produced by the program over the result of the program $\star\star$, and the evidence appearing in the program, conditioned on the chains $\mathbf{a}$. If $\mathbf{a}$ has a support environment σ, and ν is an instance of σ, $Q_{\mathbf{F}}(\star\star|\nu)$ denotes the probability of $\star\star$ and the evidence conditioned on $\mathbf{a}$ taking on the values assigned by ν.

The task of the algorithm can be formally stated as follows: We seek a function Ω from expressions ϵ and support environments σ such that

$\Omega(\epsilon, \sigma) = (S, \mathbf{F})$ where

S is a support for ϵ in σ, i.e. $S \supseteq \bigcup_{\text{instances } \nu \text{ of } \sigma} \{x : P[\epsilon, \nu](x) > 0\}$

for every instance ν of σ, $P_e[\epsilon, \nu]P[\epsilon, \nu] = Q_{\mathbf{F}}(\epsilon|\nu)$

We now present the definition of Ω for the different types of expression in the core language. For constants, we simply create a factor in which $\star$ takes on the given value with probability one, and all other values have zero probability. The support is the singleton containing the given value.

$$\Omega(s, \sigma) = \left(\{s\}, \begin{array}{|c|c|} \hline \star & \\ \hline \{s\} & \\ \hline \end{array} \right)$$

$$\begin{array}{|c|c|} \hline [\in \{s\}] & 1 \\ \hline \end{array}$$

For a variable a the simplest approach is to create a single factor with two columns, one for a and one for $\star$. The factor would contain a row for every possible value of a, assigning probability one if both a and $\star$ have the same value. All cases in which a and $\star$ have different values would have probability zero and not be listed. The support environment is useful here. We only need to enumerate those values for which the variable has positive probability, as listed in the support. Thus the size of the factor is linear in the number of values in the support of a.

Unfortunately this approach runs into trouble when we consider the fact that variables might be compound, i.e. their values might be tuples. In such a case, the possible values of the variable may be the cross product of the values of each of the components. If there are many components, or the variable represents a deeply nested data structure, there will be a huge number of

values in this cross product. The same goes for the support environment. If we map compound variables to their possible values, their supports will be huge.

A better approach is to make all the embedded components separate variables. Let the *simple chains* defined on the variable be those chains whose values are simple. The support environment only maps simple chains to domains, not compound variables. The notation $\sigma(a)$ will still be used to denote the supports of all the simple chains on a. For the translation function, we create a factor for each simple chain $a.c$, with one column for $a.c$ and one for $\star.c$. As before, we enumerate the possible values for $a.c$ as specified in the support, and create a row for each value x asserting that if both take the same value x, the probability is one.

$$\Omega(a,\sigma) = \left(\sigma(a), \left\{ \begin{array}{|c|c|c|} \hline a.c & \star.c & \\ \hline \{x_1^c, ...x_n^c\} & \{x_1^c, ...x_n^c\} & \\ \hline [\in \{x_1^c\}] & [\in \{x_1^c\}] & 1 \\ \hline & \cdots & \\ \hline [\in \{x_n^c\}] & [\in \{x_n^c\}] & 1 \\ \hline \end{array} : c \text{ is a simple chain on } a \right\} \right)$$

where $x_1^c, ..., x_n^c = \sigma(a.c)$

It should be clear that the values of a and c are equal if and only if the values of all simple chains on a and c are equal, so the probability distribution produced by the two approaches are the same. The second approach may yield an exponential reduction in the size of the factors. Whether or not this translates into savings in inference depends on whether variable elimination has to join the factors later.

To define the translation function for tuples, we will need a notion of substituting one chain for another. The *substitution* of a chain for another chain in a third chain, denoted $subst\langle c, d\rangle(e)$, is either $c.e'$, if $e = d.e'$, or e, otherwise. The notation $subst\langle c, d\rangle$ will also be used for factors and sets of factors, meaning substituting c for d in every chain in the keys of the factors. We will also use $subst\langle\{c_1, ..., c_n\}, \{d_1, ..., d_n\}\rangle$ to indicate the substitution of each c_i for d_i, respectively.

Now the translation for tuples is straightforward. Since all the component expressions generate their values independently given the environment, we can produce factors for each of the components independently. All we have to do is substitute $\star.a$ for $\star$ in component a. For the support, the support of the

entire expression is the cross-product of the supports of the components, but as above, we don't represent it explicitly, but rather maintain the supports of the simple chains.

$$\Omega(\langle a_1 = \epsilon_1, ..., a_n = \epsilon_n\rangle, \sigma) = \Big(\langle a_1 : S_1, ..., a_n : S_n\rangle, \bigcup_{i=1}^{n} subst\langle \star.a_i, \star\rangle(\mathbf{F}_i) \Big)$$
$$\text{where} (S_i, \mathbf{F}_i) = \Omega(\epsilon_i, \sigma)$$

The translation for `if` expressions requires the notion of a *conditional factor*. Intuitively, making a factor conditional on a chain taking a given value, with its given domain, means saying that the factor only matters when the chain takes on the specified value; when it takes on any other value in its domain the factor is immaterial. This is achieved by creating a new factor as follows. The key of the new factor is the key of the original factor together with the chain. (We assume that the chain was not mentioned by the original factor.) The domains of all chains other than the given chain are the same as in the original factor, and the domain of the new chain is as specified. The new factor consists of a row for every row in the original factor, and one additional row. For every row in the original factor, we create a new row with the same associated probability, in which we have the added test saying that the given chain takes on the given value. This has the effect of saying that the row only holds when the chain has the given value. In the additional row, the given chain can take anything except the given value, while the chains in the original factor can take on any value. This is saying that when the given chain takes any other value, anything goes for the chains in the original factor. Thus the factor becomes the identity when the given chain takes on a value other than the given value, which is another way of saying that the factor does not matter. Formally, we define a conditional factor as follows.

$$cond\langle c, T, x\rangle((\mathbf{c}^1, \mathbf{S}^1, \mathbf{r}^1)) = (\mathbf{c}, \mathbf{S}, \{(\phi_i, p_i^1) : (\phi_i^1, p_i^1) \in \mathbf{r}^1\} \cup \{(\psi, 1)\})$$
$$\text{where } \mathbf{c} = \mathbf{c}^1 \cup \{c\}$$
$$\mathbf{S}(c') = \begin{cases} T & \text{if } c' = c \\ \mathbf{S}^1(c') & \text{if } c' \in \mathbf{c}^1 \end{cases}$$
$$\phi_i(c') = \begin{cases} [\in \{x\}] & \text{if } c' = c \\ \phi_i^1 c' & \text{if } c' \in \mathbf{c}^1 \end{cases}$$
$$\psi(c') = \begin{cases} [\notin \{x\}] & \text{if } c' = c \\ [\notin \emptyset] & \text{if } c' \in \mathbf{c}^1 \end{cases}$$

In table form,

$$cond\langle c, T, x\rangle \left(\begin{array}{|ccc|c|} \hline c^1 & ... & c^m & \\ \hline S^1 & ... & S^m & \\ \hline \beta_1^1 & ... & \beta_1^m & p_1 \\ & \cdots & & \\ \beta_n^1 & ... & \beta_n^m & p_n \\ \hline \end{array} \right) = \begin{array}{|c|ccc|c|} \hline c & c^1 & ... & c^m & \\ \hline T & S^1 & ... & S^m & \\ \hline [\in \{x\}] & \beta_1^1 & ... & \beta_1^m & p_1 \\ & & \cdots & & \\ [\in \{x\}] & \beta_n^1 & ... & \beta_n^m & p_n \\ [\notin \{x\}] & [\notin \emptyset] & ... & [\notin \emptyset] & 1 \\ \hline \end{array}$$

The $cond\langle c, T, x\rangle$ notation is extended to apply to a set of factors, meaning making every factor in the set conditional. We can now present the translation function for `if`. A temporary variable z is created to represent the outcome of the test. The test subexpression is solved, and z is substituted for $\star$ in its factors. Assuming that both `true` and `false` are in the support of the test, we then solve the `then` and `else` subexpressions. Now, the `then` subexpression will only be evaluated when the test is `true`, so its factors only matter when z is `true`. We therefore make the factors conditional on z taking the value `true`, with the domain $\{\texttt{false}, \texttt{true}\}$. Similarly we make the factors for the `else` subexpression conditional on z being `false`. The support of the entire expression is the union of the supports of the `then` and `else` subexpressions. (I.e., a function mapping each chain to the union of its supports in the `then` and else subexpressions.)

$$\Omega(\texttt{if } \epsilon_1 \texttt{ then } \epsilon_2 \texttt{ else } \epsilon_3, \sigma) = \left(\begin{array}{cl} & subst\langle z, \star\rangle(\mathbf{F}_1) \cup \\ S_2 \cup S_3, & cond\langle z, \{\texttt{false}, \texttt{true}\}, \texttt{true}\rangle(\mathbf{F}_2) \cup \\ & cond\langle z, \{\texttt{false}, \texttt{true}\}, \texttt{false}\rangle(\mathbf{F}_3) \end{array} \right)$$

where $(S_i, \mathbf{F}_i) = \Omega(\epsilon_i, \sigma)$

When the support of the test subexpression contains only one of `true` and `false`, we can make the computation more efficient. We only have to solve one of the consequent subexpressions, and we do not need to introduce a test variable or make the resulting factors conditional. However, we do have to keep the factors from the test subexpression, because the probability of the evidence within the test may be less than one, and we need to account for that.

With the current machinery, the translation of a `dist` expression `dist` $[p_1 : \epsilon_1, ..., p_n : \epsilon_n]$ is straightforward. We create a temporary variable z to represent

the outcome of the stochastic choice, and introduce the factor

$$G = \begin{array}{|c|c|}\hline z & \\ \hline \{1,...,n\} & \\ \hline [\in \{1\}] & p_1 \\ \hline \cdots & \\ \hline [\in \{n\}] & p_n \\ \hline \end{array}$$

We then evaluate each of the subexpressions. Now, the i-th subexpression is only relevant if the i-th outcome is chosen, so we make it conditional on z having value i.

$$\begin{aligned}&\Omega(\texttt{dist}\ [p_1 : \epsilon_1, ..., p_n : \epsilon_n], \sigma) = \\ &\qquad \Big(\cup_{i=1}^{n} S_i,\ G \cup \{ cond\langle z, \{1,...,n\}, i\rangle(\mathbf{F}_i) : i \in 1,...,n\} \Big) \\ &\text{where } (S_i, \mathbf{F}_i) = \Omega(\epsilon_i, \sigma)\end{aligned}$$

For an expression `let` $a = \epsilon_1$ `in` ϵ_2, we first solve ϵ_1. We then extend the support environment by associating a with the support of ϵ_1, and solve ϵ_2. The solution to ϵ_2 will contain references to a. After we exit the `let` expression, at some point we would like to eliminate a. Because we only eliminate temporary variables, we need to create a temporary variable z and substitute it for a in the factors for ϵ_2. Furthermore, when we solve ϵ_1 the factors will mention $\star$, but in reality the result of ϵ_1 is bound to a. Since z is replacing a, we substitute z for $\star$ in the factors for ϵ_1. The support of the entire `let` expression is simply the support of ϵ_2.

$$\begin{aligned}&\Omega(\texttt{let}\ a = \epsilon_1\ \texttt{in}\ \epsilon_2, \sigma) = \Big(S_2,\ subst\langle z, \star\rangle(\mathbf{F}_1) \cup subst\langle z, a\rangle(\mathbf{F}_2) \Big) \\ &\text{where}\quad (S_1, \mathbf{F}_1) = \Omega(\epsilon_1, \sigma) \\ &\qquad\quad (S_2, \mathbf{F}_2) = \Omega(\epsilon_2, \sigma[a/S_1])\end{aligned}$$

Defining the translation function for function definitions (`fix` expressions) reveals a subtle issue. A closure is defined to include an ordinary environment, but we are using support environments. Every possible instance of the support environment defines a different function. Trying to enumerate all these possible functions would be horrendous. Fortunately, we don't have to do that. We can create a single object containing the formal variables, the body, and the support environment. This will represent all functions in which the closure environment is an instance of the support environment.

The reason we can do that is that if different instances have different probabilities, that will be captured by the factors for the expressions defining those variables. In addition, solving the body of the function will produce factors that mention the free variables used in the function. The result of applying the function will be a distribution conditioned on the values of those free variables. Multiplying the factors for the expressions defining the free variables with the result of the function will effectively be taking the expectation of the result of the function, taken over the distribution over the free variables. This will produce exactly the same effect as if we had enumerated all the possible functions and solved each one using a particular set of values for the free variables.

$$\Omega(\texttt{fix}\ a_0(a_1, ..., a_n) = \epsilon, \sigma) = \left(\{x\}, \begin{array}{|c|c|} \hline \star & \\ \hline \{x\} & \\ \hline [\in \{x\}] & 1 \\ \hline \end{array} \right)$$

where $x = \{\text{formals: } \{a_1, ..., a_n\}, \text{body: } \epsilon, \text{env: } \sigma[a_0/\{x\}]\}$

To understand this further, consider the role the support environment is playing. It is different from the role of the environment in ordinary programming language evaluation, where we look up the values of variables in the environment and use those values. Here, when we have a variable expression, we create a factor equating the value of the variable to the value of $\star$. The support is used only to restrict the possible values that have to be enumerated. As long as the support is a superset of those values that have positive probability, the result will be correct. A larger support will resultive in a more expensive computation, but not affect the correctness. When we use the closure with the support environment, this will result in solving the function body with a support environment that includes all the possible values of the free variables. So the computation will be correct.

The support plays a key role in solving function applications (again, only in terms of making the solution more efficient). When translating $\epsilon_0(\epsilon_1, ..., \epsilon_n)$, we first solve $\epsilon_0, ..., \epsilon_n$ to obtain their factors and support. Since we may have functional uncertainty, the expression to be applied may result in multiple distinct functions. By using the support, we only have to consider those functions that actually have positive probability. For each possible function, we solve the function body, in the environment formed from extending the closure environment by binding each formal argument a_i with the support computed for ϵ_i. Similar to `let` expressions, we then need to substitute a temporary variable z_i for each formal argument a_i in the result of solving the body. We also substitute z_i for $\star$ in the result of solving ϵ_i. We also create another temporary variable z_0 and substitute it for $\star$ in the result of solving ϵ_0. Finally, the

result of solving the body of a particular function is only relevant when the function to apply is actually that function, so we make the result conditional on z_0 actually being that function. The support of a function application is the union of the supports for the different possible functions to apply.

$$\Omega(\epsilon_0(\epsilon_1, ..., \epsilon_n), \sigma) = \left(\bigcup_{x \in S_0} T_x, \begin{array}{l} \bigcup_{i=0}^{n} \mathit{subst}\langle z_i, \star \rangle(\mathbf{F}_i) \\ \bigcup_{x \in S_0} \mathit{cond}\langle z_0, S_0, x \rangle(\mathbf{G}_x) \end{array} \right)$$

where $(S_i, \mathbf{F}_i) = \Omega(\epsilon_i, \sigma)$

$$\mathbf{G}_x = \mathit{subst}\langle \{z_1, ..., z_n\}, \{x.\text{formals}_1, ..., x.\text{formals}_n\} \rangle(\mathbf{H}_x)$$

$$(T_x, \mathbf{H}_x) = \Omega(x.\text{body}, x.\text{env}[x.\text{formals}_1/S_1, ..., x.\text{formals}_n/S_n])$$

In the common case that the function to apply can only have one value, we do not need to make the result of evaluating the body conditional on that value. We do however need to keep the factors obtained from solving ϵ_0, in case the probability of observations in them is less than one.

Next, we have equality tests $\epsilon_1 == \epsilon_2$. The support in general will be {`true`, `false`} but may not always be. It is important to determine the actual support because this allows us to do short-circuiting for `if` expressions as described earlier. We first solve ϵ_1 and ϵ_2, and create temporary variables z_1 and z_2 to express their results. By assumption the program is well typed and the resulting supports have the same set of chains. The result of the equality test can only be `true` if each corresponding pair of supports overlaps. The result can only be `false` if there is some pair of corresponding supports such that the size of one of the supports is not exactly one, or the supports are different. If the test must be `true` (respectively, must be `false`), we keep the factors from evaluating the subexpressions, and add a factor asserting that $\star$ is the constant `true` (resp. `false`).

In the more general case, the test could be either `true` or `false`. Here we use a *comparison* factor to test for equality between two chains. This has two rows for each possible value of the first chain. The first row says the if the first chain is x and the second chain is x, the test is true. The second says that if the first chain is x and the second chain is anything other than x, the test is false. In table form,

$$comp\langle\{c_1, S_1\}, \{c_2, S_2\}\rangle =$$

c_1	c_2	$\star$	
S_1	S_2	$\{\texttt{false}, \texttt{true}\}$	
$[\in \{x_1\}]$	$[\in \{x_1\}]$	$[\in \{\texttt{true}\}]$	1
	...		
$[\in \{x_n\}]$	$[\in \{x_n\}]$	$[\in \{\texttt{true}\}]$	1
$[\in \{x_1\}]$	$[\notin \{x_1\}]$	$[\in \{\texttt{false}\}]$	1
	...		
$[\in \{x_n\}]$	$[\notin \{x_n\}]$	$[\in \{\texttt{false}\}]$	1

where $\{x_1, ..., x_n\} = S_1$.

This is sufficient if ϵ_1 and ϵ_2 are simple expressions, i.e. they evaluate to simple values. If however they are compound, we do not want to enumerate all the possible values of ϵ_1, for the same reason as was explained earlier for variable expressions. We therefore create separate comparison factors for each simple chain c, and make the result of each a temporary variable w_c. We then need to make the result of the test the conjunction of all these temporary variables. Conjunction is a form of causal independence [Heckerman and Breese, 1994]. It is handled here using the same decomposition as presented there. The conjunction of a set of chains is a set of factors that will be denoted by $conj\langle c_1, ..., c_n\rangle$. In the most general case,

$$\Omega(\epsilon_1 == \epsilon_2, \sigma) =$$

$$\left(\begin{array}{ll} & \{subst\langle z_1, \star\rangle(\mathbf{F}_1), subst\langle z_2, \star\rangle(\mathbf{F}_2)\} \\ \{\texttt{false}, \texttt{true}\}, & \cup conj\langle w_{c_1}, ..., w_{c_n}\rangle \\ & \bigcup_{i=1}^{n} subst\langle w_{c_i}, \star\rangle comp\langle\{z_1.c_i, S_{1_i}\}, \{z_2.c_i, S_{2_i}\}\rangle \end{array}\right)$$

where $(S_i, \mathbf{F}_i) = \Omega(\epsilon_i, \sigma)$

S_{1_i} denotes the c_i component in S_1

S_{2_i} denotes the c_i component in S_2

Lastly, we have an observation expression `obs` π `in` ϵ. This is handled very simply. The pattern π defines a set of tests on simple chains on $\star$. We say that these chains are covered by the pattern. For example, the pattern $\langle$ `a : 'h, b :` $\langle$ `c : _, d : 'j` $\rangle$ $\rangle$ defines the test $[\in \{\texttt{'h}\}]$ on $\star$`.a` and $[\in \{\texttt{'j}\}]$ on $\star$`.b.d`. For each such test β on chain c with domain S, we create a factor G_c asserting

that the value of the result satisfies the test:

$$G_c = \begin{array}{|c|c|}\hline c & \\ \hline S & \\ \hline \beta & 1 \\ \hline \end{array}$$

We also include the factors from solving ϵ.

$$\Omega(\texttt{obs}\ \pi\ \texttt{in}\ \epsilon, \nu) = \cup_{\text{chains } c \text{ covered by } \pi} G_c \cup \Omega(\epsilon, \nu)$$

5.4 Efficiency Enhancing Techniques

Two techniques that greatly improve the efficiency of IBAL's inference algorithm are memoization and structured variable elimination (SVE). With memoization, when we try to compute the solution for an expression and environment, we look in a cache to see if it has already been computed and if so reuse the solution. If not, we compute the solution and store it in the cache. In practice, memoization is only performed for function applications — this appears to be sufficient. Memoization is the foundation of dynamic programming. Using it in IBAL allows IBAL to replicate the inference algorithms of frameworks that use dynamic programming, such as stochastic context free grammars.

SVE [Pfeffer, 2000] was introduced in the context of object-oriented Bayesian networks and probabilistic relational models. In SVE, each object has an *interface* to the rest of the model, consisting of variables that influence variables in the object, and variables in the object that influence other variables. When performing inference, all variables internal to the object are eliminated, and a factor over the interface is produced and communicated to the rest of the model. A similar idea is used in IBAL. Each function has an interface, consisting of the formal variables of the function and the result of the function. After solving the body of the function and computing its factors, all variables that are internal to the function are immediately eliminated. The result is a set of factors that mention only variables in the interface.

The benefits of SVE are threefold. First, using interfaces often leads to good elimination orders for VE. If all factors were sent to the top level of the program and VE only performed at the end, it might be hard to find such a good elimination order. Second, performing elimination early keeps the number of factors present at any point in the program manageable, and reduces the cost of computing elimination orderings. Third and most important, SVE together

with memoization are a powerful combination. When we memoize with SVE, we cache not only the set of factors produced when solving an expression, but also the result of the VE process on those factors. This saves us from having to perform the same VE process many times.

5.5 On Laziness

Earlier versions of IBAL used lazy evaluation, in which function arguments and variable assignments are only computed if they are actually needed to determine the value of an expression. This has been abandoned in favor of an eager approach. As a result, the supports of variables, and the factors for the expressions defining them, are always readily available when they are needed, but sometimes they will be computed when it is not necessary to do so. A strong argument can be made for lazy evaluation [Hughes, 1989] in that it greatly increases the declarativeness of the language. For example, when creating a model defining a stochastic context free grammar, one can write the model without worrying about the fact that it can produce large or even infinite sentences. When given a particular string of finite length, only the part of the process necessary for generating the string will be evaluated. With an eager approach, the model has to be "eagerified" so that a bound on the length of the produced sentence is included in every non-terminal function. The function defining the concatenation of strings becomes more complex and ugly.

Nevertheless, lazy evaluation has been discarded for two reasons. The first is that it makes the semantics very tricky to define, and non-compositional. The meaning of a function can no longer be simply a conditional probability distribution from inputs to outputs, but depends on what part of the output is needed. More seriously, laziness severely impacts the efficiency of inference.

One particular problem with lazy evaluation is that it does not work well with memoization. The reason is that in memoizing, the arguments to a function are looked up in a cache before evaluating the function. With lazy evaluation, the arguments are not available at the time the function is first evaluated. A complex scheme involving speculative evaluation was developed to handle this problem, but it imposed great overhead on the function evaluation mechanism.

5.6 Performance

When designing a language and algorithm as general as IBAL's it is impossible to expect the same performance as hand-written code specially tailored for an application. Rather, the goal has been for IBAL to be able to handle

moderately sized problems reasonably quickly. For the most part, our results are encouraging. For example, IBAL is able to solve the PRM model from Section 3.2, with the query `perf4.exam_grade`, and the observations as shown in the example, in 0.09 seconds. [3]

IBAL does reasonably well on the blocks example from Section 3.4. It is able to solve 10 blocks over 50 time steps in 7.2 seconds, when the initial configuration has all the blocks in a tower. 20 blocks over 100 time steps takes a little under 2.5 minutes. There is an exponential growth in the number of states possible as the number of time steps increases. However this growth is very slow since most actions are illegal in most states. An initial configuration in which all the blocks are on the table is much more difficult, since many more actions are legal. Here, IBAL is able to solve 10 blocks over 10 steps in 11 seconds, but 20 time steps does not terminate within 15 minutes. A special-purpose engine would have done better, but would still be limited by the inherent difficulty of the task. This illustrates that it is easy to encode models that are too complex to reason with in IBAL, and the importance of developing approximate inference algorithms.

We tested the function call mechanism of IBAL using a noisy-or like model. The recursive decomposition is implemented using nested function calls, using the function

```
f(n) =
  if n == 0
  then false
  else dist [ 0.01 : true, 0.99 : f(n-1) ]
```

IBAL is able to compute $f(1000)$ in 0.24 seconds, and the running time is linear in n on this problem. This shows that IBAL's function call mechanism is very efficient.

We also tested the ability of IBAL's representation of factors to handle a simple case of context specific independence. This involved a Bayesian network with $n + 1$ nodes, in which n nodes were roots, and the final node depended on all n of its predecessors. The conditional probability model for the final node exhibited a good deal of context specific independence. It was defined by a model of the form

```
if x0
then dist [  p0 : true, 1 − p0 : false]
else if x1
then dist [  p1 : true, 1 − p1 : false]
...
```

[3] All times are on a Linux box with 3.2 GHz processor and 1.5GB of RAM.

else if x_{n-1}
then dist [p_{n-1} : true, $1 - p_{n-1}$: false]
else dist [p_n : true, $1 - p_n$: false]

Here IBAL was able to compute the resulting distribution in 0.744 seconds for $n = 50$, and 7.124 seconds for $n = 100$. The running time displayed cubic growth on this problem. This is to be expected. The number of variables mentioned by the largest factor is linear in n. The number of rows in the largest factor is also linear in n — note that it would have been exponential in n if a table representation had been used. Finally the number of elimination steps is also linear in n. It would be interesting to find out whether a special-purpose implementation could have performed better on this problem.

We tested IBAL's ability to keep the components of a tuple separate if they do not need to be joined during the process of computation. We defined a model in which a tuple of n fields was constructed, and then the final result depended on only one of those fields. The running time grew linearly in this case, and IBAL solved a problem with $n = 100$ in 0.12 seconds. Note that an engine with lazy evaluation would have done even better. It would have recognized that the final result only depends on one attribute, and would only have created a factor to compute that attribute, and probably achieved near constant time performance.

IBAL's representation and inference algorithm can serve as the basis for a learning system. We have implemented a parameter estimation algorithm based on Expectation-Maximization (EM) [Dempster et al., 1977] on top of the inference algorithm. When run on the PRM model with 1024 observations, it takes on average 30 minutes to reach convergence. [4] The inference time for each iteration of EM grows linearly with the number of objects and observations in the model, but the number of iterations required also grows with the number of observations, so overall performance is worse than linear. The linear growth in run time per iteration is a result of the simple connectivity structure of the model. In general, as more objects are added they might become more and more interconnected, rendering exact inference infeasible for any algorithm.

6 Related Approaches

Approaches to developing expressive frameworks for probabilistic modeling can be characterized along two dimensions: the style of the representation lan-

[4] Convergence was determined to be achieved when the difference between parameter values on successive iterations was less than 0.0001.

guage, and the type of inference algorithm used. Along the first dimension, approaches can roughly be divided into four kinds: rule-based approaches, object-based approaches, undirected languages and programming language-like approaches. The different types of inference algorithms are generally knowledge-based model construction (KBMC) of a large Bayesian network, object-based methods, recursive descent on program structure, and approximate inference algorithms such as Markov chain Monte Carlo (MCMC).

In the rule-based approaches (e.g. [Poole, 1993, Ngo and Haddawy, 1996, Kersting and de Raedt, 2000]), relationships about first-order predicates are encoded by Horn clauses with uncertainty parameters. The various clauses concerning a predicate are combined together using some combination rule. The clauses, together with the combination rules, can be used in a KBMC algorithm to define a large Bayesian network representing a joint probability distribution over all ground atoms. Any BN inference algorithm can then be used to answer queries. The KBMC approach has the advantage that standard BN algorithms with their optimizations can be applied, and independence in the domain can be taken advantage of. But constructing a single large BN loses any structure that exists between high-level objects, and the same structure may be repeated many times. Sato and Kameya [2001] present a more advanced inference method that uses a tabling procedure to avoid performing redundant computations. More recently, Poole [2003] has developed a first-order variable elimination algorithm that reasons in a lifted manner, and his algorithm has been refined by R. de Salvo Braz [2005]. From the language perspective, one drawback of the rule-based approaches is that the combination rules can be hard to understand, and make the language less modular. In addition, it lacks compositional structuring elements such as objects or functions.

A second type of language uses some concept of structural elements, which are combined together to create a model. These include object-based approaches such as PRMs [Koller and Pfeffer, 1998, Pfeffer et al., 1999] and DAPER [Heckerman et al., 2004], and network-fragment based approaches such as Laskey and Mahoney [1997]. The recent MEBN language Laskey and Costa [2005] is an example of this approach that has full first-order power. PRMs use a structured, object-based inference algorithm, while MEBNs use KBMC. While these representations are very expressive, it can be hard to use them to define new kinds of models. For example, one of the features of PRMs is structural uncertainty, i.e. uncertainty about the relational structure of the domain. One form of structural uncertainty is number uncertainty, which is uncertainty about the number of elements of some kind that are related to another element. Number uncertainty cannot be easily expressed using the basic PRM representation. When the SPOOK system Pfeffer et al. [1999] was developed, it required a completely new representation infrastructure and new inference technique implemented from scratch. With IBAL, it can be expressed simply

as a language construct and an inference algorithm is automatically obtained.

Examples of undirected frameworks include relational Markov networks [Taskar et al., 2002], which can be viewed as an undirected version of PRMs, and Markov logic networks [Richardson and Domingos, 2006], in which probability weights are attached to logical clauses. These are quite distinct from the approach in this paper in that they are not generative, and thus have quite different applicability.

The programming language approaches are diverse. BUGS [Spiegelhalter et al., 1995], which uses MCMC for inference, has the appearance of an imperative language for constructing models, but as the documentation makes clear, the meaning of a program is the constructed model, not the model construction process. The expressive power of the language is not clear from the available documentation. For example, the language does not appear to have conditionals. Stochastic logic programming [Muggleton, 2001], looks at first sight like a rule-based approach, but in fact a program does not define a probability distribution over atoms in the same way as the rule-based approaches, but rather a probability distribution over proofs.

The most closely related work to this paper is [Pless and Luger, 2001], which presents a stochastic lambda calculus. The language in that paper is quite abstract. The inference algorithm is based on recursive descent over program structure, and does not fully exploit independencies that hold in the domain.

The work in this paper originated in early work in [Koller et al., 1997]. That paper introduced the idea of representing probabilistic models as programs in a functional language. The language was simpler than the one in this paper. The inference algorithm, which was based on recursive descent, was not fully worked out. It partially exploited independence, but was equivalent to forcing a bottom up variable elimination order. It proposed using lazy evaluation and memoization, without realizing that the combination of both is very difficult. It did not use the idea of converting programs into factors and then supplying them to variable elimination.

7 Conclusion

We have presented a new approach to developing expressive high-level probabilistic representation languages. The key idea is that a model can be expressed as a program describing the generative process. The meaning of the program is the probability distribution over values generated by the process. We have presented a language named IBAL that follows this idea using the functional programming paradigm. We have presented the language frame-

work and demonstrated some of the power of the language through examples. These examples show that IBAL allows for natural specification of models, is very expressive in what it can represent, makes it possible to capture model structure, is modular and compositional, and allows high-level concepts to be expressed in a very declarative way.

In addition, we have presented a semantics for IBAL in terms of a distribution defined by a program. We have also presented a powerful inference algorithm that exploits many kinds of structure. The inference algorithm exploits independence, low-level structure of models, high-level object structure, repetition in models, and utilizes the support of expressions to limit the amount of computation that needs to be performed. The algorithm implements several techniques to keep the factors used in variable elimination small. As a result, the algorithm is quite general, and captures a number of specific frameworks. For example, it reduces to the standard algorithm for Bayesian networks, hidden Markov models, and stochastic context free grammars, when those frameworks are represented in the language. In addition, it allows these structural features to be exploited when they are found in a new framework.

There are several natural directions for future work. One is to allow continuous variables, which will greatly expand the applicability of the language. Another is to introduce approximate inference, which is quite necessary given how easy it is to create complex models for which exact inference does not work. A more specific language enhancement would be to provide a pass by reference mechanism for function applications. In the current implementation, factors are created matching entire function arguments to their values. This can be inefficient when the function arguments are large, as for example in lists. A mechanism that would allow the body of a function to refer to its arguments without having to entirely match the arguments could provide great speedup in some cases.

References

A. Becker and D. Geiger. A sufficiently fast algorithm for finding close to optimal clique trees. *Artificial Intelligence*, 125:3–17, 2001.

C. Boutilier, N. Friedman, M. Goldszmidt, and D. Koller. Context-specific independence in Bayesian networks. In *Uncertainty in Artificial Intelligence (UAI)*, 1996.

E. Charniak. *Statistical Language Learning*. MIT Press, 1993.

R. Dechter. Bucket elimination: a unifying framework for reasoning. *Artificial Intelligence*, 113(1–2):41–85, 1999.

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

D. Heckerman and J. S. Breese. A new look at causal independence. In *Uncertainty in Artificial Intelligence (UAI)*, pages 286–292, 1994.

D Heckerman, C Meek, and D Koller. Probabilistic entity-relationship models, PRMs, and plate models. In *ICML-2004 Workshop on Statistical Relational Learning*, 2004.

J. Hughes. Why functional programming matters. *The Computer Journal*, 32 (22):98–107, 1989.

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.

U. Kjaerulff. Triangulation of graph-based algorithms giving small total space. Technical report, University of Aalborg, Denmark, 1990.

D. Koller, D. McAllester, and A. Pfeffer. Effective Bayesian inference for stochastic programs. In *National Conference on Artificial Intelligence (AAAI)*, pages 740–747, 1997.

D. Koller and A. Pfeffer. Probabilistic frame-based systems. In *National Conference on Artificial Intelligence (AAAI)*, pages 580–587, 1998.

K. B. Laskey and P. Costa. Of klingons and starships: Bayesian logic for the 23rd century. In *Uncertainty in Artificial Intelligence (UAI)*, 2005.

K. B. Laskey and S. M. Mahoney. Network fragments: Representing knowledge for constructing probabilistic models. In *Uncertainty in Artificial Intelligence (UAI)*, pages 334–341, 1997.

B. Milch, B. Marthi, S. Russell, D. Sontag, D. L. Ong, and A. Kolobov. BLOG: Probabilistic models with unknown objects. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2005.

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

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

J. Pearl. *Probabilistic Reasoning in Intelligent Systems*. Morgan Kaufmann, 1988.

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

A. Pfeffer, D. Koller, B. Milch, and K. T. Takusagawa. SPOOK: A system for probabilistic object-oriented knowledge representation. In *Uncertainty in Artificial Intelligence (UAI)*, pages 541–550, 1999.

D. Pless and G. Luger. Toward general analysis of recursive probability models. In *Uncertainty in Artificial Intelligence (UAI)*, pages 429–436, 2001.

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

D. Poole. First-order probabilistic inference. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2003.

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

D. Roth R. de Salvo Braz, E. Amir. Lifted first-order probabilistic inference.

In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2005.

L.R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2), 1989.

M Richardson and P Domingos. Markov logic networks. *Machine Learning*, 62(1–2):107–136, 2006.

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

K. Shoikhet and D. Geiger. A practical algorithm for finding optimal triangulations. In *National Conference on Artificial Intelligence (AAAI)*, 1997.

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.

R. E. Tarjan and M. Yannakakis. Simple linear time algorithms to test chordality of graphs, test acyclicty of graphs, and selectively reduce acyclic hypergraphs. *SIAM Journal on Computing*, 13:566–579, 1984.

B. Taskar, P. Abbeel, and D. Koller. Discriminative probabilistic models for relational data. In *Uncertainty in Artificial Intelligence (UAI)*, 2002.

N. L. Zhang and D. Poole. A simple approach to Bayesian network computations. In *Tenth Biennial Canadian Artificial Intelligence Conference*, 1994.