IBAL: A Probabilistic Rational Programming Language

Avi Pfeffer
Division of Engineering and Applied Sciences
Harvard University
avi@eecs.harvard.edu

Abstract

In a rational programming language, a program specifies a situation faced by an agent; evaluating the program amounts to computing what a rational agent would believe or do in the situation. This paper presents IBAL, a rational programming language for probabilistic and decision-theoretic agents. IBAL provides a rich declarative language for describing probabilistic models. The expression language allows the description of arbitrarily complex generative models. In addition, IBAL's observation language makes it possible to express and compose rejective models that result from conditioning on the observations. IBAL also integrates Bayesian parameter estimation and decision-theoretic utility maximization thoroughly into the framework. All these are packaged together into a programming language that has a rich type system and built-in extensibility. This paper presents a detailed account of the syntax and semantics of IBAL, as well as an overview of the implementation.

1 Introduction

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

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

IBAL makes four main contributions. The first is a highly expressive language for representing probability models, that is significantly more expressive than previous languages. Second, IBAL integrates a language for probabilistic modeling, Bayesian learning and decision theory under a single coherent semantic framework. Third, it provides a unified inference engine for solving reasoning, learning and utility maximization problems, that generalizes algorithms for many standard kinds of models. Finally, IBAL is packaged together into a usable programming language with features such as a strong type system and built-in extensibility.

IBAL is designed with three kinds of users in mind. The first is the system modeler, who may not be an expert in probabilistic reasoning. For this type of user the basics of the language should be reasonably easy to learn, and it should be fairly easy to come up with decent models for many domains. This kind of user will benefit greatly from a good selection of libraries implementing standard kinds of models. Also, a good default inference algorithm is needed that can be expected to do reasonably well on a large number of models.

The second kind of user is a modeling expert, who understands well the inference algorithms for probabilistic reasoning. For the expert user, the language should provide the power to carefully tweak the model being used, and to control what inference algorithm is used to evaluate different parts of the model.

The third kind of user is the AI researcher, who may want to introduce new kinds of probabilistic models and new inference algorithms. IBAL makes it easy to introduce new models via libraries, and the implementation framework provides support for building new algorithms and extending the inference capabilities of the system.

Space limitations prevent a full description of both the language and the implementation in this paper. The next section provides a fairly detailed account of the language and its semantics. Section 4 presents an extended example, showing how the various components of IBAL can be used together to provide a declarative implementation of a fairly complex decision-theoretic agent. Section 5 presents an overview of the main features of the IBAL implementation.

2 Genealogy and Related Work

The most direct presucrsor to IBAL is the language of [Koller *et al.*, 1997], hereinafter referred to as KMP97. KMP97 is a Lisp-like language extended with a `flip` construct to describe random events. IBAL extends KMP97 in a number of powerful ways. IBAL's basic definition and expression language for describing generative probabilistic models is similar to KMP97, but significantly richer, particularly in its use of higher-order functions. In addition, IBAL uses observations to condition distributions, which allows it to easily describe a much richer class of models. IBAL also integrates decisions and learning into the framework, which are not provided by KMP97.

IBAL is also indebted to recent work integrating probabilistic and first-order representation languages. Two recent strands in that direction are relational probability models [Pfeffer *et al.*, 1999], and stochastic logic programs [Muggleton, 2000].

Other projects have tried to integrate an expressive modeling language with at least some aspect of learning or decision theory. Another project similar in spirit to IBAL is DTGolog [Boutilier *et al.*, 2000], which integrates decision theory into the Golog action cacululus. It is based on logic-programming rather than the functional programming framework IBAL uses. Also, its roots can be traced back to Markov decision processes, while IBAL's roots are in Bayesian networks. As a result, the two systems have quite a different approach to inference; for example, there is no notion of variable elimination in DTGolog. Also, DTGolog does not integrate learning into the framework.

[McAllester, 2000] presents a language based on KMP97 for describing decision problems and the policies used by agents, and for calculating the expected utilities of the agents. Unlike IBAL, there is no attempt to solve the decision problem and compute the optimal policies. [Cumby and Roth, 2000] integrates learning and reasoning in an expressive, compositional language, but one that is not probabilistic.

3 The IBAL Language

IBAL provides a declarative language for describing probability distributions, parameter estimation problems and utility maximization problems. The top level language component in IBAL is a *block*. A block consists of a sequence of *declarations*. There are a number of kinds of declarations, including *definitions* stating how values of things are stochastically generated; *observations* stating that some property holds of generated values; *priors* describing the prior probability distributions over learnable parameters; *decisions* describing the decisions that an agent makes and the information it has; *rewards* describing the rewards an agent receives; and *pragmatics*, containing control knowledge on how to perform inference in a block. I will describe each of these in turn. For the sake of presentation I will present the semantics of the language incrementally, as I discuss each kind of declaration. However, the discussions of semantics are somewhat technical, and can be skipped on first reading.

3.1 Definitions

A *definition* states how the value associated with a name is generated. A *name* is a symbol such as x; a *chain* is a sequence of names such as $x.y.z$. A definition has the form $x = e$, where e is an *expression* with one of the forms:

(Constant)	$'s$ where s is a symbol
(Chain)	σ
(Function)	$\lambda(x_1, \ldots, x_n) \rightarrow e$
(Conditional)	`if` $e_1 \models \pi$ `then` e_2 `else` e_3 where π stands for a pattern, discussed below
(Dist)	`dist` $[p_1{:}e_1, \ldots, p_n{:}e_n]$ where the p_i are probabilities summing to 1
(Block)	$\{b\}$ where b is a block
(Application)	$e_0(e_1, \ldots, e_n)$

A *pattern* defines a condition that may be satisfied by a value. The notation $e \models \pi$ stands for the predicate that is true iff the value of expression e satisfies the pattern π. A pattern may specify equality to a constant, a chain, it may be the negation of a pattern, or the conjunction or disjunction of two patterns.

In addition to the above expression syntax, the language provides plenty of syntactic sugar, such as case statements and easier syntax for defining functions. There is also a type system, discussion of which will be omitted for lack of space.

The intuitive meaning of a definition $x = e$ is that it defines a stochastic experiment generating the value of x. Consider a simple example:

```
fair()    = dist [0.5 : 'h, 0.5 : 't]
biased() = dist [0.9 : 'h, 0.1 : 't]
pick()    = dist [0.5 : fair, 0.5 : biased]
coin       = pick()
x            = { y = coin() ; z = coin() }
```

Intuitively, `fair` and `biased` are functions that return either `'h` or `'t` with appropriate probabilities. The function `pick` is a higher-order function that returns either the fair or biased function. The value of `coin` is generated by applying pick; it is either `fair` or `biased`. The expression defining `x` is a block expression; `x` is a data structure with components `y` and `z`, both generated by applying the value of `coin`. Chains `x.y` and `x.z` are mutually dependent on whether `coin` is `fair` or `biased`, but they are conditionally independent given `coin`.

IBAL inherits from KMP97 the idea of using *laziness* to allow infinitely long experiments to be defined, only some of whose results may be needed. For example:

```
real() = { first = dist [0.5 : 'zero, 0.5 :'one]
           rest = real() }
less_than_half = real.first |= 'zero
```

Here, `real` defines a uniform distribution over real numbers between 0 and 1, represented by their binary expansion. Executing `real` involves an infinite recursion, but only the first bit is needed to determine the value of `less_than_half`, which is the value of a Boolean predicate that looks only at `real.first`. We can think of executing `real` lazily, to get the single bit needed.

I will now make these intuitions precise. There are four kinds of values in IBAL: (1) Symbols such as heads and true; (2) A special *undefined* value, denoted by $\perp$; (3) *Complex values* that consist of an *environment*, which maps names to values; (4) *Closures*, denoted $(x_1,\ldots,x_n) \xrightarrow{\epsilon} e$ consisting of formal paramaters $x_1,\ldots,x_n$, body e and environment ϵ.

A chain can be viewed as representing a function on values. Formally, if σ is a chain and v a value, we define $\sigma(v)$ as follows: If σ is empty, $\sigma(v) = v$. Otherwise, if v is not complex, $\sigma(v) = \perp$. Otherwise, let σ be $x.\sigma'$, and v' be the value associated with x in v; then $\sigma(v) = \sigma'(v')$. To determine whether a value satisfies a pattern, we need an environment assigning values to the variables appearing in the pattern. Formally, we define $v \models^\epsilon \pi$, meaning that v satisfies π in ϵ, by $v \models^\epsilon {'s}$ if v is the symbol s; $v \models^\epsilon \sigma$, if $\sigma(\epsilon) = v$ and $v \neq \perp$; and $v \models^\epsilon \neg\pi$ if $v \not\models^\epsilon \pi$ and $v \neq \perp$. $v \models^\epsilon \pi$ for conjunctive or disjunctive π is defined in the obvious way. Note that $\perp$ cannot satisfy any pattern.

In order to generate the value of an expression e, we need an environment ϵ binding the free variables of e, and a source of randomness, which is provided by an infinite sequence ρ of i.i.d. real numbers $\in [0,1)$. Formally, we will define a function $\langle e\rangle^{\epsilon,\rho}$, meaning the value generated for e in environment ϵ, given random choices as in ρ. The notation ρ_i^n is the subsequence of ρ of elements with index congruent to i modulo n. This device allows us to split ρ into multiple independent subsequences. $\mathrm{Hd}[\rho]$ and $\mathrm{Tl}[\rho]$ indicate the head and tail of ρ. For constant, chain, function, conditional and dist expressions, $\langle e\rangle^{\epsilon,\rho}$ is defined by:

$$
\begin{array}{lcl}
\langle {'s}\rangle^{\epsilon,\rho} & = & s \\
\langle \lambda(x_1,\ldots,x_n) \to e\rangle^{\epsilon,\rho} & = & (x_1,\ldots,x_n) \xrightarrow{\epsilon} e \\
\langle \sigma\rangle^{\epsilon,\rho} & = & \sigma(\epsilon) \\
\left\langle \begin{array}{l} \texttt{if } e_1 \models \pi \\ \quad\texttt{then } e_2 \\ \quad\texttt{else } e_3 \end{array} \right\rangle^{\epsilon,\rho} & = & \left\{ \begin{array}{ll} \langle e_2\rangle^{\epsilon,\rho_2^2} & \text{if } v \models^\epsilon \pi \\ \langle e_3\rangle^{\epsilon,\rho_2^2} & \text{if } v \models^\epsilon \neg\pi \\ \perp & \text{if } v = \perp \end{array} \right. \\
 & & \text{where } v = \langle e_1\rangle^{\epsilon,\rho_1^2} \\
\langle \texttt{dist}[p_1{:}e_1,\ldots,p_n{:}e_n]\rangle^{\epsilon,\rho} & = & \langle e_i\rangle^{\epsilon,\mathrm{Tl}[\rho]} \\
\multicolumn{3}{c}{\text{where } \sum_{j=1}^{i-1} p_i \le \mathrm{Hd}[\rho] \le \sum_{j=1}^{i} p_i}
\end{array}
$$

The value of a block expression is complex, and a definition $x_i = e_i$ within the block results in the x_i component mapping to the value of e_i. Each definition is evaluated in an environment including bindings for names appearing previously in the block. Furthermore, if e_i is a lambda expression, the binding for x_i is also added to the environment in which e_i is evaluated, so that x_i is bound in the resulting closure. This allows recursive functions to be defined. Formally:

$$
\begin{array}{l}
\langle \{x_1 = e_1; \ldots; x_n = e_n\}\rangle^{\epsilon,\rho} = \{x_1{:}v_1; \ldots; x_n : v_n\} \\
\quad \text{where} \quad v_i = \langle e_i\rangle^{\epsilon_i,\rho_i^n} \\
\quad\quad \text{and} \quad \epsilon_i = \left\{ \begin{array}{ll} \epsilon[x_1 \ldots x_i / v_1 \ldots v_i] & \text{if } e_i \text{ is a lambda} \\ \epsilon[x_1 \ldots x_{i-1} / v_1 \ldots v_{i-1}] & \text{otherwise} \end{array} \right.
\end{array}
$$

The final case to define is function application. To determine the value of $e_0(e_1,\ldots,e_n)$, we first compute the value of e_0. If it is not a closure, the result of the application is undefined. Otherwise we evaluate the body in the environment formed by extending the closure environment by binding each formal parameter x_i to the value of e_i. Formally:

$$
\langle e_0(e_1,\ldots,e_n)\rangle^{\epsilon,\rho} = \left\{ \begin{array}{l} \langle e'\rangle^{\epsilon'[x_1\ldots x_n/v_1\ldots v_n],\rho_{n+2}^{n+2}} \\ \quad \text{if } \langle e_0\rangle^{\epsilon,\rho_{n+1}^{n+2}} = (x_1,\ldots,x_n) \xrightarrow{\epsilon'} e' \\ \perp \text{ otherwise} \end{array} \right.
$$
$$
\text{where } v_i = \langle e_i\rangle^{\epsilon,\rho_i^{n+2}}
$$

Note that the preceding definitions elegantly take care of the issue of infinite experiments with finite observations, such as the `less_than_half` example above, without needing to make explicit use of laziness in the semantics. The rule for generating the value of an expression only uses the chains that appear in it. Furthermore, a block expression always returns a complex value.

The above semantics is an *operational semantics*, showing how a program consisting of definitions defines a random experiment for generating values. We also provide a *denotational semantics* in terms of a probability measure over values. The underlying probability space consists of countable sequences of i.i.d. real numbers generated uniformly from [0,1). A program $\mathcal{I}$ defines a function from sequences to values, by $\mathcal{I}(\rho) = \langle\{b\}\rangle^{\epsilon_0,\rho}$, where ϵ_0 is the empty environment. If R is a measurable set of sequences, and S is the image of R under $\mathcal{I}$, then S is measurable and $\Pr(S) = \Pr(R)$.

Natural properties of values that we would like to talk about are in fact measurable. For example, any property of the form $\sigma(v) = s$ is measurable. We can see this by defining a *depth-bounded* evaluation function $\langle e\rangle_n^{\epsilon,\rho}$. Its definition is the same as above, except that $\langle\rangle_0$ is always $\perp$, and $\langle\rangle_n$ uses $\langle\rangle_{n-1}$ for evaluating the body of function applications. In other words, values that require a recursion to a depth greater than n will be undefined. Now, a program defines a sequence of functions $\mathcal{I}_i(\rho) = \langle\{b\}\rangle_i^{\epsilon_0,\rho}$. It is clear that the set $R_i = \{\rho : \sigma(\mathcal{I}_i(\rho)) = s\}$ is measurable since it only requires looking at a finite subsequence of ρ to determine if it is in R_i. Therefore the set $R = \{\rho : \sigma(\mathcal{I}(\rho)) = s\} = \cup_i R_i$ is measurable. Furthermore, the above argument also suggests an anytime approximation algorithm for computing $\Pr(S)$. Since the R_i are non-decreasing, and their union is R, the probabilities of the R_i are a non-decreasing sequence whose limit is $\Pr(R) = \Pr(S)$.

3.2 Observations

The language described so far is similar to that of KMP97, albeit with a richer syntax and type system, and a more refined semantics. It can express many common models, such as Bayesian networks, relational probability models, stochastic logic programs, hidden Markov models, dynamic Bayesian networks and stochastic context free grammars. All these models are *generative* in nature, defining an experiment that stochastically generates values for variables. The richness of the model is encoded in the way the values are generated.

Another flavor of probability model is a *rejective* model. In a rejective model, the data is generated by a very simple process, e.g. uniformly, but data that fails to satisfy certain constraints may be rejected. The richness of the model is encoded in the rejection process. A good example of a rejective model is a *product of experts (POE)* [Hinton, 2000]. In a POE, a datum x is generated uniformly, and then passed to a set of probabilistic experts. Each expert i accepts x with some probability $p_i(x)$ that depends on a property of x. The data is accepted only if all experts accept it. The probability of any datum x is proportional to $\prod_i p_i(x)$.

IBAL is able to express rejective models by making observations an integral part of the language. An *observation* is a declaration of the form $e \models \pi$. Recall that this is the syntax used for Boolean predicates. An observation is simply a statement that a certain predicate is true.

Thus, for example, the general schematic form of a POE model is expressed as follows.

```
generate()      =    . . .  (* produce a uniform datum *)
x               =    generate()
(* for each expert i, the following code *)
expert_i(x)     =    . . .  (* return 'accept or 'reject *)
expert_i(x)     |=   'accept
```

Another kind of model that can be expressed using observation declarations is a *Markov random field (MRF)*. An MRF is an undirected analogue of a Bayesian network, but it can also be viewed as a type of POE. An interesting effect is at work here. IBAL's expression language defines directed, generative models, but the observation language implements the undirected notion of a constraint. As a result, IBAL is able to express both directed and undirected models, and combinations of the two. It is important to stress that observations are an integral part of the language, and not something pasted onto a model after the fact in order to condition it. They can occur within blocks and functions, and therefore they can be composed together, just like generative definitions. All the power of a modular, functional language is thereby extended to rejective models. Of course, IBAL also allows rejective models to be combined with generative models. For example, one natural way to build a language model is to use a stochastic context free grammar as the initial generator of sentences, and then use probabilistic constraints to express global properties like agreement and sentence length.

In defining the semantics of observations in IBAL, one subtle point must be stressed. An observation in a block can only condition variables defined within the block, not free variables. As far as a containing block is concerned, the definition of a contained block is considered to be a black box. It simply defines a distribution over the value of the block, given values for the free variables. The containing block need not concern itself with whether this distribution is defined generatively or rejectively. Failure to enforce this rule would be a serious violation of modularity.

We get the right effect simply by modifying the definition of $\langle\{b\}\rangle^{\epsilon,\rho}$ for the case of block expressions. Now, in addition to defining values for each of the components of the block, it will also make sure that all the observations in the block are satisfied. If they are not, the generated values for the block are rejected, and the process is repeated. The resulting distribution defined by the block is conditioned on the observations being satisfied. It is, of course, possible to define a set of observations that fail with probability 1, in which case the attempt to generate a value for the block will go on for ever. In that case, the value of the block is $\perp$. Note that because the rejection/repetition process is contained within the block itself, only the value of the block itself is conditioned by the observations, not the free variables.

3.3 Learning

Observations provide the basis for integrating learning, in the form of Bayesian parameter estimation, into the IBAL framework. Unknown probability parameters are specified using *prior* declarations, which have the form `learn` $x =$ `dirichlet` $[\alpha_1 : e_1, \ldots, \alpha_n : e_n]$.

A prior declaration of this form achieves two things. First, it defines a probabilistic parameter $\theta^x = \theta_1^x, \ldots, \theta_n^x$, and specifies a Dirichlet prior over the parameter. The α_i are positive real numbers, specifying the the hyperparameters of the Dirichlet. Second, a prior declaration also creates a definition for x, equivalent to $x =$ `dist` $[\theta_1^x : e_1, \ldots, \theta_n^x : e_n]$.

We can view an IBAL program with prior declarations as specifying a joint model, that defines a joint probability distribution over the model parameters and the value returned by the program. Observations condition the joint model in the standard Bayesian way. Let us refine the coins example from earlier by adding priors and observations.

```
fair()      =    { result = dist [0.5 : 'h, 0.5 : 't] }
biased()    =    { learn result = dirichlet [90 : 'h, 10 : 't] }
pick()      =    { learn result = dirichlet [1 : fair, 1 : biased] }
coin        =    pick().result
x           =    { y = coin().result ; z = coin().result }
x.y         |=   'h
```

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

With its learning component, IBAL is able to do parameter estimation for many common models, including hidden Markov models, stochastic context free grammars and probabilistic relational models. Furthermore, learning in IBAL is not just "added on" to the probabilistic representation language, but is thoroughly integrated into the language. As a result, the benefits of compositionality and modularity are obtained for representing learning tasks. In particular, IBAL is good at representing a cumulative learning framework, in which smaller models are learned and then used as components of larger learning problems. Just as observations only condition values within their scope, they are only used to learn about model parameters within their scope. Thus a compositional learning process can be specified by providing a nested scope containing all the data and parameters for a learning subproblem, and a containing scope that uses the results of the subproblem.

In defining the semantics of learning in IBAL, we need to get a subtle point correct. If a prior declaration appears in the body of a function, the same parameter values should be used every time the function is applied. This is fundamental to learning — different observations of the same function are all observations about the same parameter! This means that in terms of the generative semantics, the parameter value is not returned by each application of the function, but rather it is generated when the function is defined, and stored as part of the closure representing the value of the function object.

To achieve this effect, we make the following definition. The parameters *directly inside* a block are the parameters of prior declarations defined in the block, that are not nested in the body of a lambda expression. In the generative process, the values of the parameters directly inside the body of a lambda are generated at the time the closure is created, and these values are used for all future applications of the closure.

The remainder of the semantics stays basically the same as before. When a definition resulting from a prior declaration is encountered, the relevant parameter values are looked up in the environment and used to choose which branch is taken, in the same way as for a `dist` expression.

3.4 Decisions and Utilities

The representation of decision problems in IBAL is geared towards two popular models: influence diagrams (IDs) and Markov decision problems (MDPs). IBAL can easily represent these and other models, including various kinds of structured MDPs. A decision declaration in IBAL has the form `choose` x `from` $s_1, \ldots, s_n$ `given` $\sigma_1, \ldots, \sigma_m$. This specifies the name x of the decision variable, its range $s_1, \ldots, s_n$, and the information available to the decision maker, the chains $\sigma_1, \ldots, \sigma_m$ (called the *informational parents* of x). A block may contain multiple decision declarations, in which case we enforce the no-forgetting rule of IDs, that the informational parents of later decisions always include those of earlier decisions, as well as the decisions themselves. A reward declaration is either `receive case` e `of` $[\pi_1\colon r_1, \ldots, \pi_n\colon r_n]$ or `receive` $\alpha\sigma$. The first form states that the reward depends on the value of e, and it is the real number r_i associated with the first pattern π_i that the value satisfies. The second states that the reward is α times the reward of σ, where α is a positive real number. A block may have multiple reward declarations, in which case the total reward is the sum of the individual rewards. For example, a typical MDP is schematically represented as follows:

```
MDP(s) = {   (* takes current state as argument *)
   transition(s,a) = ...   (* returns next state *)
   reward(s,a) = { reward case (s,a) of [('s1,'a1) : 3, ... ] }
   choose a from a1,a2 given s
   next_state = transition(s,a)
   current_reward = reward(s,a)
   future_reward = MDP(next_state)
   reward 1 current_reward
   reward 0.9 future_reward }
```

Three points must be made about the representation of decision problems in IBAL. The first is that each block constitutes a distinct decision problem. A block with decisions is viewed as identifying an implicit agent who makes the decisions and receives the rewards mentioned in the block itself. Decisions and rewards in nested blocks implement the notion of delegation; decisions in the nested block do not consider their effects on the containing block. The reason for enforcing this interpretation is similar to the reason observations don't condition values outside their scope; the alternative would result in a serious loss of modularity. If an agent in a nested block had to be concerned about rewards in the calling block, the decisions for the nested block would no longer be determined solely by its free variables. Therefore a program calling the nested block could no longer treat it as a black box.

The second point to clarify is that it is assumed that the values of free variables are always known to an agent making the decisions in a block. Once again, failure to enforce this restriction would result in a modularity problem. If the agent does not know the values of the free variables, it must know the distribution over those values in order to make a rational decision. But then, the distribution defined by a block depends not only on the values of the free variables, but on their distribution. This is an unfortunate restriction, since it prevents IBAL from being capable of representing POMDPs. The situation can perhaps be salvaged, by borrowing the idea of belief state from POMDPs, and adding it as an extra implicit input to the block.

The third point is the relationship between observations and decisions in a block. We assume that all observations are known to the decision maker, even if they appear lexically subsequent to the decision. The reason is that the observations are viewed simply as part of the definition of the probability distribution over the block's values, and we make the assumption that the decision maker knows the correct distribution. A result of this assumption is that we disallow observations statement to mention variables that depend (directly or indirectly) on the decisions in a block. This restriction prevents the semantics of learning and decisions from interfering with each other.

In defining the semantics, we begin with utilities. With every value v we associate a utility $U(v)$. For non-complex values $U(v) = 0$. For complex values $U(v)$ is determined by the reward declarations in the block in which v was created. Recall that a complex value is created by $\langle\{b\}\rangle^{\epsilon,\rho}$. We extend the definition of $\langle\{b\}\rangle^{\epsilon,\rho}$ so that it also produces $U(v)$ after producing v. We set $U(v)$ to be the sum over reward declarations R in b of $U_R(v)$, defined as follows. If R is `reward case` σ `of` $\pi_1 : r_1, \ldots, \pi_n : r_n$,

$$U_R(v) = \begin{cases} r_i & \text{if } \sigma(v) \models^{\epsilon'} \pi_i \text{ and } \sigma(v) \not\models^{\epsilon'} \pi_j \text{ for } j < i \\ 0 & \text{if } \sigma(v) \not\models^{\epsilon'} \pi_i \text{ for } i = 1, \ldots, n \end{cases}$$

where ϵ' is the environment resulting from extending ϵ with the bindings in v. If R is `reward` $\alpha\sigma$, $U_R(v) = \alpha \cdot U(\sigma(v))$.

Now for decisions. A *strategy* d for block b is a decision $d_{D,w}$ for each decision statement D and each value w of the informational parents of D. Given an environment ϵ in which to evaluate b, and a particular strategy d, we define a function $U(b)_d^{\epsilon,\rho}$, whose meaning is the utility for b, given strategy d, bindings of free variables in ϵ, and random choices specified by ρ. Holding ϵ, b and d fixed, this function is a random variable on the space of sequences. Taking the expectation of this function over sequences produces the expected utility; the optimal strategy for b is then the one that maximizes this expected utility. The definition of $\langle\{b\}\rangle^{\epsilon,\rho}$ now proceeds by first choosing $d^* = \arg\max_d E_\rho[U(b)_d^{\epsilon,\rho}]$,and then processing the block as before, using d^* to choose the values of each decision variable given its informational parents.

3.5 Pragmatics

The IBAL language allows a wide variety of different kinds of models to be expressed. There is no single best inference algorithm to use for all models. While IBAL implements a good default algorithm that should work well in many situations, it also allows the possibility of using other methods. The programmer can specify control knowledge on how to solve a program in the form of *pragmatic declarations*. These could either specify what algorithm to use, or details of how to use a particular algorithm, such as specific elimination orderings for variable elimination.

Because pragmatic declarations are part of a block, they only specify how to do inference in that block and its nested blocks. Furthermore, pragmatic declarations in a nested block can override those in a containing block. These features allow fine, modular, control over how inference is done, allowing complex models to be built in which different inference methods are used for different components.

It is anticipated that pragmatic declarations will often be used by library designers. The user of a library is shielded from thinking about how the library models will be solved. This modularity of inference methods may turn out to be as important as modularity of representation in building complex probabilistic agents.

4 Example

In this section, I illustrate how a declarative high-level representation language that unifies probabilistic reasoning, decision theory and parameter estimation can simplify and integrate the implementation of sophisticated rational agents. Consider the task of implementing an automated receptionist agent. The job of the agent is to receive spoken requests over the phone and to respond to them appropriately. Requests include asking for directions and talking to a particular person; responses include giving directions, connecting to an extension, and asking the caller to repeat the request.

In a decision-theoretic design for this agent, the agent receives a utility based on how well its response matches the actual request of the user. Of course, the agent does not observe the actual user request, but only a sequence of signals. In the decision model, there will be a prior distribution over requests and a conditional distribution of signals given requests.

With existing tools, it is difficult to implement this decision-theoretic design using a single, coherent model. Instead, a typical implementation might consist of several components, each of which is probabilistic, but which are stitched together in an ad-hoc manner. There will typically be a speech-recognition component that determines the likely words that generated the received signal; a language model that can determine the probability that a particular request generated a particular sentence; and a high-level influence diagram for deciding what to do. The results of the speech-recognizer and request generator are fed into the language model. The results of the language model are fed into the influence diagram. The overall result may not be a coherent probabilistic model. Furthermore, it may be hard to tailor the components for their use in this application. For example, getting the speech recognizer to recognize unusual names of individuals in the company might require an extra engineering effort.

With IBAL, the whole application can be described using a single declarative model. At a high-level, the generative model consists of three steps: a request generation function, a function that generates sentences based on requests, and a function that generates phoneme sequences from sentences. For the decision-making component, the agent is given the phoneme sequence, and chooses a response. The agent's utility is determined from the request and the response. The high-level code looks like this:

```
request = make_request()
sentence = make_sentence(request)
phonemes = make_phonemes(sentence)
choose response.type from GiveDirections, Connect, Repeat
choose response.who from fred, wilma
receive match(request, response)
```

I will now describe, in turn, the three generative steps and the utility computation. The function `make_request` produces a request. A request has a `type` field, whose value is either `GetDirections` or `Talk`. A `Talk` request also has a `who` field, whose value is a person. A person is a complex value with three fields: `name`, `title` and `extension`. In a typical programming language, the type of `name` would be string, because what we typically want to do with names is compare them, read them and print them. In this application, however, the most important thing about a name is how it tends to be pronounced. Therefore the type of `name` is a function that stochastically generates a phoneme sequence. The same is true for `title`. The `make_request` function is as follows:

```
make_request() = {
  learn who = [1 : fred, 1 : wilma]
  learn type = [20 : GetDirections; 80 : Talk]
}
```

Generating the request involves choosing two things: the type of the request, and the person, if the request is to talk to a person. Both choices are made learnable. The agent has lots of past experience about what kinds of requests tend to be generated. These are expressed by observation statements as follows:

```
r1 = make_request()
r1.type |= GetDirections
r2 = make_request()
r2.type |= Talk
r2.who |= wilma
...
```

The `make_sentence` function produces a sentence from a request. A sentence is a list of words, each of which is a function that stochastically produces a list of phonemes when activated. Here I illustrate with an extremely simple sentence generation model; in a real application, a more complex grammar model might be used. Even this simple example illustrates the power of passing around data structures that contain functions in fields. The sentence generator simply slots the given person's name or title in the appropriate place. ($[x_1; \ldots; x_n]$ is the list containing $x_1, \ldots, x_n$, and @ is the list concatenation operator.)

```
make_sentence(request) =
  if request.type ⊨ GetDirections
    then [can;you;give;me;directions]
    else [connect;me;to] @
      [dist [0.5:request.who.name, 0.5:request.who.title]]
```

The `make_phonemes` function takes a sentence and produces a list of phonemes. Since each word in the sentence is a function that generates phoneme sequences, `make_phonemes` just executes each of the words in the sentence, and concatenates the results. The individual word models would probably be hidden Markov Models, and may be part of a speech recognition library. Unusual names of people may not be in the generic hMM library, but the library may provide a function that creates a new, trainable hMM that can be learned for a particular name.

```
make_phonemes(sentence) =
  if sentence.is_empty
    then []
    else head(sentence)() @ make_phonemes(tail(sentence))
```

Finally, the utility model takes the request and response, and produces a utility of 1 if the request matches the response. If the response is RepeatPlease, the utility is 0. An incorrect response has utility -5.

```
match(request,response) = {
  correct =
    (request.type ⊨ GetDirections ∧
      response.type ⊨ GiveDirections) ∨
    (request.type ⊨ Talk ∧
      response.type ⊨ Connect ∧
      request.who ⊨ response.who)
  receive
    if correct
      then 1
      else if response.type ⊨ Repeat
        then 0
        else -5
}
```

5 Implementation Overview

The inference tasks in IBAL are to estimate the learnable parameters, to compute the utility maximizing decisions, and to solve for the conditional distribution over various chains, given values for other chains. The tasks are accomplished in that order. Since decisions cannot influence observations, they are irrelevant to the learning task. Once the parameters have been learned, the expected utility for any strategy can be computed and the decision task can be solved. Once the decisions have been fixed, the distribution over all values is known and the probabilistic reasoning task can be solved.

IBAL is implemented in Objective CAML, a variant of ML. The design of the implementation is divided into four parts. The first component is a *frontend* consisting of a parser, type checker, and translator, whose job is to produce code in *shallow form*. In shallow form, nested subexpressions are replaced by chains, and only simple patterns are allowed. The second component is a set of *support modules*. These include a ubiquitous polymorphic container type implementing efficient maps from chains to values of any kind, as well as a general implementation of *events*, which are measurable sets of values, and *factors*, which are measurable functions from values to elements of a ring. Events and factors support a very generalized version of the variable elimination algorithm.

The third component implements the *main line* of inference. The main line begins with code in shallow form, and proceeds through a sequence of steps. The first step is *domain generation* in which the support $D(\sigma)$ of each chain used in the program is computed. Domain generation uses rules such as the following: for a definition $w = \texttt{dist}\ [p_1 : \sigma_1, \ldots, p_n : \sigma_n]$, $D(w)$ is the union of the $D(\sigma_i)$. Domain generation also uses observations to restrict the domains. Domain generation may be followed by an optional *constraint propagation* step, which further restricts the domains, by propagating observations back through the definitions in the program. Constraint propagation may be worthwhile since it is cheaper than the full-scale variable elimination process performed later.

The second step is to compute the *needed chains* that are actually relevant to answering a query. These include all chains that influence query variables or observed variables, directly or indirectly. This computation step is particularly important in recursive models such as stochastic grammars. It deduces, for example, that in order to determine whether the first word of a sentence is "the", only the first non-terminal need be expanded at any stage (assuming there are no ϵ productions).

The key step in the main line, *factor production*, converts each definition in the program into a set of factors. The goal is to produce factors that are as simple as possible; dummy variables may be introduced to achieve this. For example, for a definition $x = y$, suppose x and y are complex, with $x.a$ and $x.b$ needed. Two separate factors will be produced, one enforcing that $x.a$ and $y.a$ are equal, and the other for $x.b$ and $y.b$. For a definition $x = \texttt{dist}\ [p_1 : \sigma_1, \ldots, p_n : \sigma_n]$, a dummy variable d is introduced, to represent which branch is actually taken. The dummy variable serves to separate the σ_i from each other; without the dummy variable, they would all have to appear in the same factor. A set of factors is produced for each branch i, saying that if d takes the value i, then the components of x and σ_i must have the same value, but if d takes any other value this branch is irrelevant and there is no constraint on x and σ_i. An additional factor saying that d takes value 1 with probability p_1, …, n with probability p_n is also produced.

The final solution step is *variable elimination*, as is commonly used for BN inference. The set of factors computed by factor production represents a sum of products expression. All variables except for query variables and free variables in a block are eliminated from the factors, and the result is the conditional probability of the query variables given the free variables. All solution steps except for the final variable elimination are recursive; a recursive call is used to process nested blocks and function applications.

The final component of the implementation is the *glue* that holds everything together. Each of the solution steps is turned into a dynamic programming algorithm, where all recursive calls are looked up in a cache before solving them explicitly. Also, an iterative deepening strategy is used to provide an anytime approximation algorithm for queries that may not terminate. These recursion strategies are implemented in a generic, modular way. For example, a `dynamic` function is provided that takes a recursive algorithm and produces a dynamic programming version of the same algorithm. The glue is also responsible for handling pragmatics.

The glue is implemented using a programming technique in which a recursive function f takes a special argument g. When f recurses, rather than calling itself directly, it calls g. Thus f is a function with a "hole", which needs to be filled in. All the recursion strategies can be implemented as ways of filling in the hole. The simplest way to fill in the hole is to plug f into itself, which gives the basic recursion pattern. But another possibility is to define a function `cached`(f) which looks for a result in a cache before calling f, calls f if the result is not found, and stores the result of calling f in the cache. If `cached`(f) is plugged into itself, the result is a dynamic programming algorithm based on f. Similarly, `depth-bounded`(n, d, f) produces a version of f that stops recursing after a recursion depth of n, returning d instead. Recursion strategies can be composed using the method of functions with holes. This provides a nice, modular way to deal with multiple inference methods. Each inference method is invoked by a pragmatics handler that recognizes a particular type of pragmatic declaration. The pragmatic handlers can be installed on top of each other, so that every recursive call results in the all the handlers being checked. The inference methods themselves do not need to know what handlers are used for the recursive calls.

All three inference problems use dynamic programming with the main line of inference described above. Decision making is based on ID algorithms, using backward induction to solve the decisions in a block from the bottom up. For MDPs, the algorithm reduces to value iteration (because of the dynamic programming component) with reachability analysis (because of the domain generation). For the probabilistic reasoning task, all reasoning is directed towards a particular query; pruning of the problem to those variables required for the query is accomplished by computing the needed chains. For probabilistic reasoning tasks, the algorithm reduces to regular variable elimination for standard BNs, while the dynamic programming makes it equivalent to forward-backward for HMMs and the inside algorithm for SCFGs. Finally, parameter estimation uses a general version of the EM algorithm to compute maximum a-posteriori parameter values. This is only an approximation to the true Bayesian posterior, of course. The E step consists of using the main inference line to compute how many times each branch was taken for each prior declaration. These are the expected sufficient statistics. The M step combines the expected sufficient statistics with the Dirichlet hyperparameters to produce new parameter values that have maximum posterior probability given the expected sufficient statistics. Again, the combination of variable elimination with dynamic programming results in the algorithm reducing to the typical EM used for BNs, Baum-Welch for HMMs, and the outside algorithm for SCFGs.

I do not claim that IBAL will be an efficient inference algorithm for all applications. Some problems are inherently hard, or require special methods, and a variety of algorithms is needed. I do believe that IBAL integrates a number of widely used methods, and thereby provides a good default algorithm. Furthermore, the support modules and glue make it easier to extend the system with alternative methods.

6 Conclusion and Future Work

IBAL is a rich declarative programming language for describing probabilistic models, decision theoretic situations, and Bayesian parameter estimation problems. This paper has presented a syntax and semantics for the language, and an overview of an implementation with probabilistic reasoning, utility maximization and parameter learning.

Future work on IBAL is divided into implementation enhancements that are practically useful but not too theoretically challenging, and more significant extensions to the expressive power of the language. Plans in the first category include implementing a variety of approximate solution methods; providing libraries to implement common kinds of models, and useful structures such as sets of objects; and extending the type system to include algebraic data types and polymorphic types, modeled on ML.

One of the main future tasks in the second category, is to support the learning of model structure as well as parameters. In keeping with the philosophy that everything that can be done in IBAL should be thoroughly integrated, this will require a much richer language with which to express priors, including priors over structure. Another important extension is to provide a way to describe how an IBAL agent interacts with its environment; in other words, to provide a declarative description of the I/O capaboilities of the agent. A final task is to allow multiple agents into the IBAL framework. Not only would this allow IBAL to be used for modeling game-theoretic situations, it would also provide a way to describe agents' models of other agents. While these tasks present a range of interesting and challenging issues, IBAL provides a good foundation with which to begin tackling them.

References

[Boutilier *et al.*, 2000] C. Boutilier, R. Reiter, M. Soutchanski, and S. Thrun. Decision-theoretic, high-level agent programming in the situation calculus. In *AAAI*, 2000.

[Cumby and Roth, 2000] C. Cumby and D. Roth. Relational representations that facilitate learning. In *KR*, 2000.

[Hinton, 2000] G. Hinton. Training products of experts by minimizing contrastive divergence. Technical report, Gatsby Computational Neuroscience Unit, 2000.

[Koller *et al.*, 1997] D. Koller, D. McAllester, and A. Pfeffer. Effective Bayesian inference for stochastic programs. In *AAAI*, 1997.

[McAllester, 2000] D. McAllester. Bellman equations for stochastic programs. Revision of talk at LPNMR-99, 2000.

[Muggleton, 2000] S. Muggleton. Stochastic logic programs. *Journal of Logic Programming*, 2000. Accepted subject to revision.

[Pfeffer *et al.*, 1999] A. Pfeffer, D. Koller, B. Milch, and K.T. Takusagawa. SPOOK: A system for probabilistic object-oriented knowledge representation. In *UAI*, 1999.