THE UNIVERSITY of EDINBURGH

Edinburgh Research Explorer

Tabular: A Schema-driven Probabilistic Programming Language

Citation for published version:
Gordon, AD, Graepel, T, Rolland, N, Russo, C, Borgstrom, J & Guiver, J 2014, Tabular: A Schema-driven Probabilistic Programming Language. in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM SIGPLAN Notices, no. 1, vol. 49, ACM, New York, NY, USA, pp. 321-334, 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California, United States, 22/01/14. https://doi.org/10.1145/2535838.2535850

Digital Object Identifier (DOI):
10.1145/2535838.2535850

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

OPEN ACCESS

Download date: 25. Apr. 2024

Tabular: A Schema-Driven Probabilistic Programming Language

Andrew D. Gordon
Microsoft Research and University of Edinburgh

Thore Graepel
Microsoft Research

Nicolas Rolland
Microsoft Research

Claudio Russo
Microsoft Research

Johannes Borgström
Uppsala University

John Guiver
Microsoft Research

Abstract

We propose a new kind of probabilistic programming language for machine learning. We write programs simply by annotating existing relational schemas with probabilistic model expressions. We describe a detailed design of our language, Tabular, complete with formal semantics and type system. A rich series of examples illustrates the expressiveness of Tabular. We report an implementation, and show evidence of the succinctness of our notation relative to current best practice. Finally, we describe and verify a transformation of Tabular schemas so as to predict missing values in a concrete database. The ability to query for missing values provides a uniform interface to a wide variety of tasks, including classification, clustering, recommendation, and ranking.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Specialized application languages; I.2.6 [*Artificial Intelligence*]: Learning—Parameter Learning

Keywords Bayesian reasoning; machine learning; model-learner pattern; probabilistic programming; relational data

1. Introduction

The core idea of this paper is to write probabilistic models by annotating relational schemas. We illustrate this idea on a database for recording outcomes of a two-player game without draws.

Players	
Name	**string**

Matches	
Player1	**link**(Players)
Player2	**link**(Players)
Win1	**bool**

In this *concrete schema*, we have a Players table with column Name, and a Matches table, with columns Player1, Player2, and Win1 ("Player 1 wins"). As well as scalar types such as **bool** or **string**, a column may have a type such as **link**(Players), which means the column holds integer foreign keys to the Players table. (For simplicity, we assume that every table has a single-column primary key ID, a common case in practice. We also assume that in a table with n rows the keys are integers normalized to lie in the range $0..n-1$; thus, we omit the primary key column from schemas.)

To illustrate some of the key ideas of Tabular, we consider the TrueSkill model (Herbrich et al. 2006), which is deployed at cloud-scale to make selections of players of roughly equal skill as opponents in online gaming. In this model, each player has an underlying numeric skill Skill, players' performances in a match are noisy copies of their skills, and each match is won by the player with the greater performance.

Players			
Name	**string**	**input**	
Skill	**real**	**latent**	Gaussian(25.0,0.01)
Matches			
Player1	**link**(Players)	**input**	
Player2	**link**(Players)	**input**	
Perf1	**real**	**latent**	Gaussian(Player1.Skill,1.0)
Perf2	**real**	**latent**	Gaussian(Player2.Skill,1.0)
Win1	**bool**	**output**	Perf1 > Perf2

Although its starting point is the underlying concrete schema, a Tabular schema may contain additional *latent columns*, which contain random variables to help model concrete data. In our example, the Players table has a latent column Skill, containing a numeric skill for each player, while the Matches table has latent columns Perf1 and Perf2, containing the performances of the two players in the match.

So that a schema defines a probability distribution over database instances, we annotate columns with probabilistic *model expressions*, which define distributions over entries in the column. Model expressions allow predictions to be made for the values of associated columns. Our example shows three sorts of annotated column:

(1) A concrete column marked as an *output* has a model expression that predicts values of the column. For example, the Win1 column is an output; its model expression indicates the winner is the player with the greater performance. The model expression can be applied to predict a future match outcome based on skills learnt from training data.

(2) A concrete column marked as an *input* is used to condition the probabilistic model, but has no model expression and cannot be predicted by the model. For example, the Player1 column in the Matches table is an input; it is used to characterize a match but is not considered to be uncertain.

(3) Finally, a column marked as *latent* is an auxiliary column, not present in the concrete database, whose model expression forms part of the model, and can be predicted. For example, the Skill column has a model expression indicating each entry is drawn from a Gaussian distribution with mean 25 and precision 0.01.

A Tabular program divides the columns of the concrete database into input and output columns, and determines a probabilistic model that predicts the output columns given the input columns. If all the cells in a concrete column have values we say the column is *observed*, but otherwise, when there are missing values, we say it is *observable*.

We consider two forms of inference. In both forms, input columns are observed. In *query-by-latent-column*, we assume that output columns are observed—we have data for each cell in the column—and the task is to predict the latent columns. Towards the end of the paper, in Section 7, we also consider *query-by-missing-value*, where output columns are observable, and the task is to predict the missing values in output columns.

Query-by-Latent-Column Given a table of players and a table listing the outcomes of matches between those players, TrueSkill infers a numeric skill for each player, used for matchmaking. Consider the following tables of players and matches.

Players

ID	Name
0	`"Alice"`
1	`"Bob"`
2	`"Cynthia"`

Matches

ID	Player1	Player2	Win1
0	0	1	**false**
1	1	2	**false**

Initially, TrueSkill assigns the same uncertain skill prior to each player. Given data showing that player 0 has been beaten by player 1, who in turn has been beaten by player 2, TrueSkill infers posterior skill distributions reflecting the likely ranking player 0 $<$ player 1 $<$ player 2.

The *query-by-latent-column* problem for Tabular is to determine the probability distribution over latent databases for a given schema, given a concrete database. In theory, the latent database is a joint distribution over all latent columns of the database. In a practical implementation, we consider only the marginals (projections) of each of the variables in the latent database. In particular, for the TrueSkill schema, conditioned on the concrete database above, the marginal representation of the distribution over latent databases consists of the following tables.

PlayersLatent

ID	Skill
0	Gaussian(22.51, 1.45)
1	Gaussian(25.22, 1.53)
2	Gaussian(27.93, 1.45)

MatchesLatent

ID	Perf1	Perf2
0	Gaussian(22.49, 1.11)	Gaussian(25.25, 1.14)
1	Gaussian(25.25, 1.14)	Gaussian(27.96, 1.11)

The distribution over the latent database can be stored in the same relational store as the original concrete database, joined with the concrete tables. While Tabular is specific to the domain of specifying probabilistic models for relational data, users are free to deploy whatever programming or query notation is appropriate to preprocess the data into relational form and to postprocess the results of inference.

Query-by-Missing-Value In this mode, we use tables with missing values in observed columns as queries. For example, the following amounts to a query asking how likely it is that player 2 would beat player 0, to help decide on placing a bet.

Matches

ID	Player1	Player2	Win1
3	2	0	?

The result of such a query might be the following, indicating there is an 85% chance player 2 will beat player 0.

MatchesQueryLatent

ID	Win1
3	Bernoulli(0.85)

1.1 A Schema-Driven Recipe for Probabilistic Modelling

In designing Tabular, we have in mind *data enthusiasts* (Hanrahan 2012), the large class of end users who wish to model and learn from their data, who have some knowledge of probability distributions and database schemas, but who are not necessarily professional programmers.

Tabular supports the following recipe for modelling data.

(1) Start with the schema (such as the Players and Matches tables).

(2) Add latent columns (Skill, Perf1 and Perf2).

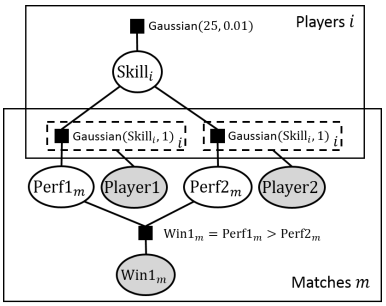
(3) Write probabilistic models for latent and observed columns (skills have a prior, performances are noisy copies of skills, the player with the highest performance wins).

(4a) Learn latent columns and table parameters from complete data (we learn players' skills from a dataset of match outcomes).

(4b) Or predict missing values from partially-observed data (we predict a future match outcome based on a row (p1,p2,?)).

There is more to the whole cycle of learning from data—such as gathering and preprocessing data, and visualizing and interpreting results—but the recipe above addresses a crucial component.

Models as Factor Graphs Factor graphs are a standard class of probabilistic graphical models of data, with many applications (Koller and Friedman 2009). Having modelled data with a factor graph, one can apply a range of inference algorithms to infer properties of the data or make predictions. The TrueSkill model was originally expressed as a factor graph such as the one below, in typical "plates and gates" notation.



The circular nodes of the graph represent random variables, and the black squares are factors relating random variables. The large enclosing boxes labelled "Players i" and "Matches m" are known as *plates*, and indicate that the enclosed subgraphs are to be replicated. The two dotted boxes are known as *gates* (Minka and Winn 2008), and indicate choices governed by an incoming edge. (Gates are essential for many models, and we include gates here to illustrate their use, but they are not strictly necessary in TrueSkill, as the player links are not stochastic. If we had uncertainty about the players, the gate would be essential for inference.) The nodes for some random variables are shaded to indicate they are *observed*, while unshaded variables are *latent*. Together with exact factor annotations, factor graphs represent joint probability distributions.

Like many visual notations, factor graphs become awkward as models become complex. Instead, we turn to probabilistic programming languages, where models are code, random variables are program variables, factors are primitive operations, plates are loops, and gates are conditionals or switches. BUGS (Gilks et al. 1994; Lunn et al. 2013) is the most popular example, and there is much current interest, witness the wiki `probabilistic-programming`.

`org`. In this paper, we create models with a direct interpretation as factor graphs by writing schema annotations in a high-level probabilistic language.

1.2 Innovations in the Design of Tabular

By using the relational modelling of the data encoded in the concrete schema, we write models succinctly because each table description implicitly defines a loop (a plate) over its rows. Moreover, we save our user the trouble of writing code to transfer data and results between language and database. The main conceptual innovations in Tabular are:

(1) Annotations on a relational schema so as to construct a graphical model, with *input*, *output*, and *latent* columns.

(2) A grammar of model expressions to stipulate the models for latent and output columns, with the semantics of tables and schemas given as models assembled compositionally from the models for individual columns.

(3) *Query-by-latent-column*: infer latent columns from the concrete database, given input columns and fully-observed output columns.

(4) *Query-by-missing-value*: infer missing values in output columns, given input columns and partially-observed output columns.

1.3 Technical Contributions and Evaluations

We present the detailed syntax and type system of Tabular, and semantics by translation to a core probabilistic calculus, Fun. Theorem 1 (Translation Preserves Typing) asserts that the semantics respects the Tabular type system. Theorem 2 asserts that a certain factor graph, expressed in Fun, correctly implements query-by-latent-column.

We describe an implementation of Tabular using Infer.NET, based on our semantics. To test Tabular in practice, we reimplement a series of factor-graph models for psychometric data first performed using Infer.NET directly (Bachrach et al. 2012), with essentially the same results. Theorem 3 (Query-by-Missing-Value) justifies a transformation on Tabular schemas that implements query-by-missing-value in terms of query-by-latent-column.

In the case of machine learning on data held in a database, an advantage of schema-driven probabilistic programming over probabilistic forms of conventional programming languages is that there is no need to map between database schemas and programming language types. Since the inference results in steps (4a) and (4b) of our recipe above work by reading tables from a relational store and writing the results of inference back into the relational store, we are not dependent on any particular language-based representation of data or data-binding. Hence, a Tabular programmer avoids the usual impedance mismatch problem between databases and programming languages (Maier 1987) and is free to pre-process data and post-process inference results using whatever data access technology is appropriate, such as a spreadsheet, or a scripting language, or a full programming language.

1.4 Structure of the Paper

Section 2 recalls Fun (Borgström et al. 2013), a typed first-order fragment of the stochastic lambda-calculus, that serves as our notation for graphical models. We also recall the model-learner pattern (Gordon et al. 2013), a way of structuring Bayesian models compositionally, and the basis of our semantics for Tabular. Section 3 introduces Tabular's column annotations, grammar of model expressions, generative process for tables and schemas, and query-by-latent-column, by example. Section 4 defines a formal semantics for Tabular, based on translation to the model-learner pattern. Our semantics treats model expressions, tables, and whole schemas as model combinators. Section 5 describes our implementation, based on the formal semantics. Section 6 outlines a substantial case study. Section 7 considers query-by-missing-value, where Tabular predicts missing values in output columns. We discuss examples and show how to transform Tabular schemas so as to reduce query-by-missing-value to query-by-latent-column. Section 8 describes related work and Section 9 concludes.

An appendix includes additional examples, benchmark results, and screenshots of our implementation.

2. Fun and the Model-Learner Pattern

2.1 Fun, Probabilistic Programming for Factor Graphs

We use a version of the core calculus Fun (Borgström et al. 2013) with arrays of deterministic size, but without a conditioning operation (**observe**) within expressions. This version of Fun can be seen as a first-order subset of the stochastic lambda-calculus (Ramsey and Pfeffer 2002); it is akin also to HANSEI (Kiselyov and Shan 2009). Borgström et al. (2013) show how to translate Fun to the Infer.NET input format, a probabilistic imperative language, with much of the work being to eliminate records; in a similar way, we could translate Fun to BUGS (Gilks et al. 1994; Lunn et al. 2013). Fun expressions have a semantics in the probability monad, but also have a direct interpretation using factor graphs.

We have scalar types **bool**, **int**, and **real**, record types (that are constructed from field typings), and array types. Let $\textbf{string} = \textbf{int}[]$ and $\textbf{vector} = \textbf{real}[]$ and $\textbf{matrix} = \textbf{vector}[]$. Let c range over the field names, s range over constants of base type, and let $\mathrm{ty}(s) = T$ mean that constant s has type T.

Types and Values (Scalars, Records, Arrays): T, V

$S ::= \textbf{bool} \mid \textbf{int} \mid \textbf{real}$	scalar type
$T, U ::= S \mid \{RT\} \mid T[]$	type
$RT ::= \varnothing \mid c : T; RT$	field typings
$V ::= s \mid \{c_1 = V_1; \ldots; c_n = V_n\} \mid [V_1, \ldots, V_n]$	

Expressions of Fun: E

$E, F ::=$	expression
$x \mid s$	variable, constant
$\textbf{if } E \textbf{ then } F_1 \textbf{ else } F_2$	if-then-else
$\{R\} \mid E.c$	record literal, projection
$[E_1, \ldots, E_n] \mid E[F]$	array literal, lookup
$[\textbf{for } x < E \rightarrow F]$	for-loop (scope of index x is F)
$\textbf{let } x = E \textbf{ in } F$	let (scope of x is F)
$g(E_1, \ldots, E_n)$	primitive g with arity n
$D(E_1, \ldots, E_n)$	distribution D with arity n
$R ::= \varnothing \mid c = E; R$	field bindings

We write $\mathrm{fv}(\phi)$ for the set of variables occurring free in a phrase of syntax ϕ, such as an expression E, and identify syntax up to consistent renaming of bound variables. We sometimes use tuples $(E_1, \ldots, E_n)$ and tuple types $T_1 * \cdots * T_n$ below: they stand for the corresponding records and record types with numeric field names 1, 2, …, n. We write $\textbf{fst } E$ for $E.1$ and $\textbf{snd } E$ for $E.2$. The *empty record* $\{\}$ represents a void or unit value. We write $\{c_1 : T_1; \ldots; c_n : T_n\}$ for a concrete record type, and thus $\{\}$ for the empty record type; $\{c_1 = E_1; \ldots; c_n = E_n\}$, for a concrete record term; and use the comprehension syntax $\{c_i : T_i\}^{i \in 1..n}$ and $\{c_i = E_i\}^{i \in 1..n}$ to index the components of a record type or term (when ordering matters) or $\{c : T_c\}^{c \in C}$ and $\{c = E_c\}^{c \in C}$ (where C is a *set* of field names) when ordering is irrelevant. Field typings and field bindings are just association lists; we sometimes use $RT_1; RT_2$ to denote the *concatenation* of field typings RT_1 and RT_2, and $R_1; R_2$ for the concatenation of field bindings. We implicitly identify record types up

to re-ordering of field typings. We assume a collection of total deterministic functions g, including arithmetic and logical operators. We also assume families D of standard probability distributions, including, for example, the following.

Distributions: $D : (x_1 : T_1; \ldots; x_n : T_n) \to T$

Bernoulli : (bias : **real**) → **bool**	
Gaussian : (mean : **real**, precision : **real**) → **real**	
Beta : (a : **real**, b : **real**) → **real**	
Gamma : (shape : **real**, scale : **real**) → **real**	
DirichletSymmetric : (length : **int**, alpha : **real**) → **vector**	
Discrete : (probs : **vector**) → **int**	
DiscreteUniform : (range : **int**) → **int**	
VectorGaussian : (mean : **vector**, covariance : **matrix**) → **vector**	

2.2 Semi-Observed Models

We explain the semantics of Tabular by translating to Bayesian models encoded using Fun expressions. We consider a Bayesian model to be a probabilistic function, from some *input* to some *output*, that is governed by a *parameter*, itself generated probabilistically from a deterministic *hyperparameter*. Our semantics is compositional: the model of a whole schema is assembled from models of tables, which themselves are composed from models of rows, assembled from models of individual cells. This formulation follows Gordon et al. (2013), with two refinements. First, when we apply a model to data, the model output is *semi-observed*, that is, each output is a pair consisting of an observed component (like a game outcome in TrueSkill) plus an unobserved latent component (like a performance in TrueSkill). Second, the hyperparameter is passed to the sampling distribution $\mathsf{Gen}(h,w,x)$ as well as to the parameter distribution $\mathsf{Prior}(h)$ for convenient model building. Passing the hyperparameter to the sampling distribution is a convenience when building models, but does not alter the expressivity of the abstraction; if the hyperparameter is not passed explicitly to the output distribution we can always pass it implicitly as an extra component of the parameter.

Notation for Bayesian Models:

Hyper	E_h	default hyperparameter (E_h deterministic)
$\mathsf{Prior}(h)$	E_w	distribution over parameter (given h)
$\mathsf{Gen}(h,w,x)$	E_{yz}	distribution over output (given h, w, and x)

(Hyperparameters and parameters both determine the distribution of outputs given an input; the difference is that we specify our uncertain knowledge of parameters (but not hyperparameters) using the prior distribution, so that our uncertainty about parameters (but not hyperparameters) is reduced by conditioning on data.)

For example, here is a model for linear regression, that is, the task of fitting a straight line to data points. This example illustrates the informal notation for Fun expressions used in Section 3. For instance, we write $a \sim \mathsf{Gaussian}(h.\mu_A, 1)$ to mean that random variable a is distributed according to $\mathsf{Gaussian}(h.\mu_A, 1)$. We write $x := E$ to indicate that x is the value of deterministic expression E.

Linear Regression: (Illustrative of informal notation for Fun)

Hyper	The record $\{\mu_A = 0; \mu_B = 0\}$.
$\mathsf{Prior}(h)$	The record $\{A = a; B = b\}$ where $a \sim \mathsf{Gaussian}(h.\mu_A, 1)$ and $b \sim \mathsf{Gaussian}(h.\mu_B, 1)$.
$\mathsf{Gen}(h,w,x)$	The pair (y,z) where $z := (w.A) * x + w.B$ and $y \sim \mathsf{Gaussian}(z, 1)$.

In our formal semantics for Tabular, we use a compact notation $P ::= \langle E_h, (h)E_w, (h,w,x)E_{yz} \rangle$ for a model. Our regression example is written in compact notation as follows.

$$\begin{array}{l}
\langle \{\mu_A = 0; \mu_B = 0\}, \\
\quad (h)\textbf{let } a = \mathsf{Gaussian}(h.\mu_A, 1) \textbf{ in} \\
\qquad \textbf{let } b = \mathsf{Gaussian}(h.\mu_B, 1) \textbf{ in } \{A = a; B = b\}, \\
\quad (h,w,x)\textbf{let } z = (w.A) * x + w.B \textbf{ in let } y = \mathsf{Gaussian}(z, 1) \textbf{ in } (y,z) \rangle
\end{array}$$

We use variable x for the input, y for the observed output, z for the latent output, w for the parameter, and h for the hyperparameter. These variables range over Fun values and hence may be scalars, but may also be compound structures such as whole databases.

2.3 Databases as Fun Values

We view a database as a record $\{t_1 = B_1; \ldots; t_n = B_n\}$ holding (relational) tables $B_1, \ldots, B_n$ named $t_1, \ldots, t_n$. A table B is an array $[r_1, \ldots, r_m]$ of rows, where each row is a record $r_i = \{c_1 = V_1; \ldots; c_n = V_n\}$, where $c_1, \ldots, c_n$ are the columns of the table, and $V_1, \ldots, V_n$ are the items in the column for that row. (We view a table as an array so that a primary key is simply an index into the array, and omit primary keys from rows.)

The column annotations in a Tabular schema partition a whole database into a pair $d = (d_x, d_y)$ where d_x is the *input database*, with the input columns of each table, and d_y is the *observed database*, with the observed columns of each table. (For each table, the numbers of rows in the input and observed databases must match.)

The *latent database* d_z is a database with just the latent columns of the schema, and the *database parameter* V_w is a record holding parameters for each table.

The purpose of *query-by-latent-column* is to predict the database parameter and latent database from the input and observed databases.

Distributions Induced by a Semi-Observed Model In later sections, we define the semantics of a Tabular schema as a model P. In general, a model P defines several probability distributions:

- *Prior* $p(w \mid h)$ is $w \sim P.\mathsf{Prior}(h)$.
- *Full sampling* $p(y,z \mid h,w,x)$ is $y,z \sim P.\mathsf{Gen}(h,w,x)$.
- *Sampling distribution* $p(y \mid h,w,x)$ is $\int p(y,z \mid h,w,x)dz$.
- *Predictive distribution* $p(y \mid x,h)$ is $\int p(y \mid h,w,x)p(w \mid h)dw$.

Training data for a model consists of a pair $d = (d_x, d_y)$ where d_y is the observed output given input d_x. In our case, d_x is the input database and d_y is the observed database. Conditioned on such data $d = (d_x, d_y)$ we obtain posterior distributions:

- *Posterior* $p(w \mid d,h) = \frac{p(d_y|h,w,d_x)p(w|h)}{p(d_y|d_x,h)}$.
- *Posterior latent* $p(z \mid d,h) = \int \frac{p(d_y,z|h,w,d_x)}{p(d_y|h,w,d_x)} p(w \mid d,h)\,dw$.

(The term $p(d_y \mid d_x, h)$ is known as the *evidence* for the model, used later in our comparison of different models on the same dataset).

Given $d = (d_x, d_y)$, the *semantics of query-by-latent-column* is to compute the posterior $p(w \mid d,h)$ on the database parameter, and the posterior latent distribution $p(z \mid d,h)$ on the latent database.

3. Tabular, By Example

3.1 Tabular and the Generative Process for Tables

As usual, a relational schema confers structure on a database. A schema $\mathbb{S}$ is an ordered list of tables, named $t_1, \ldots, t_n$, each of which has a table descriptor $\mathbb{T}$, that is itself an ordered list of typed columns, named $c_1, \ldots, c_n$. The key concept of Tabular is to place an annotation A on each column so as to define a probabilistic model for the relational schema.

We present first a core version of Tabular, where the model expressions M on columns are simply Fun expressions E.

Tabular Schemas, Tables and Annotations: $\mathbb{S}$, $\mathbb{T}$, A

$\mathbb{S} ::= \varnothing \mid (t \mapsto \mathbb{T})\mathbb{S}$		(database) schema
$\mathbb{T} ::= \varnothing \mid (c \mapsto A : T)\mathbb{T}$		table descriptor
$A ::=$		annotation
$\quad$ **hyper**(E)		$\quad$ hyperparameter
$\quad$ **param**(M)		$\quad$ parameter
$\quad$ **input**		$\quad$ input
$\quad$ **output**(M)		$\quad$ output
$\quad$ **latent**(M)		$\quad$ latent
$M ::= E$	(to be completed)	model expression

The types T on concrete columns are typically scalars, but our semantics allows these types to be arbitrary. The Tabular syntax for types and expressions slightly extends Fun syntax with features to find the sizes of tables and to dereference foreign keys.

Additional Types and Expressions of Tabular Fun: T, E

$T ::= \cdots \mid$ **link**(t)	type
$E ::= \cdots \mid$ **sizeof**$(t) \mid (E :$ **link**$(t)).c$	expression

The expression **sizeof**(t) returns the number of rows in table t. The expression $(E :$ **link**$(t)).c$ returns the item in column c of the row in table t keyed by the integer E. In the common case when E is a column c_k annotated with type **link**(t), we write $c_k.c$ as a shorthand for $(c_k :$ **link**$(t)).c$. Values of type **link**(t) are integers serving as foreign keys to the table t. For simplicity, our type system treats each type **link**(t) as a synonym for **int**.

Generative Process for Tables A table descriptor $\mathbb{T}$ is a function from the concrete table holding the **input** and **output** columns, to the *predictive table*, which additionally holds the **latent** columns. The descriptor defines a generative process to produce (1) the hyperparameters and parameters of the table, and (2) the output and latent columns of the table, by a loop over the rows of the table.

In step (1), outside the loop over the data, we process the annotations in turn to define the hyperparameters and parameters, ignoring the input, output, and latent annotations.

- $c \mapsto$ **hyper**(E) defines c as the deterministic expression E.
- $c \mapsto$ **param**(E) samples c from probabilistic expression E.

In step (2), a loop over each row of the concrete table, we process the annotations in turn to sample independently each row of the predictive table, with items for each of the input, output, and latent columns.

- $c \mapsto$ **input** copies c from the input row.
- $c \mapsto$ **output**(E) samples c from probabilistic expression E.
- $c \mapsto$ **latent**(E) samples c from probabilistic expression E.

In step (2), inside the data loop, we ignore the hyperparameter and parameter annotations, although expressions may depend on the variables defined in step (1) outside the loop.

A schema $\mathbb{S}$ describes a generative process to produce (1) the hyperparameters and parameters of each table, and (2) the predictive table for each concrete table. Tables and columns are lexically scoped in sequence, although the variables bound in step (1) cannot refer to variables bound later in step (2).

Later on, we formalize the generative processes for tables and schemas using our model notation; step (1) corresponds to the Hyper and Prior parts, while step (2) corresponds to the Gen part.

Example: Conjugate Bernoulli This standard model is used to generate random bits with a probability distribution that is itself random; it is a key ingredient of mixture models.

CoinFlips			
alpha	**int**	**hyper**	1
beta	**int**	**hyper**	1
Bias	**real**	**param**	Beta(alpha,beta)
Coin	**bool**	**output**	Bernoulli(Bias)

In step (1) of the generative process, we define both alpha and beta as 1, and sample Bias from the distribution Beta(1,1), the uniform distribution on the unit interval. In step (2), we generate each row of the table by sampling the Coin variable from the distribution Bernoulli(Bias) on **bool**, which returns **true** with probability Bias. Overall, we sample the shared parameter Bias, whereas we sample each output Coin independently for each row.

A concrete database for this schema is simply one table with a single column Coin containing Booleans. Inference computes the distribution of the Bias parameter.

3.2 Distributions with Conjugate Priors

In Bayesian theory, the Beta distribution over the parameter of the Bernoulli distribution is a particular case of a *conjugate prior*. It is convenient for efficient inference to choose a prior that is conjugate to a sampling distribution. Hence, we define *primitive models* for various standard sampling distributions and conjugate priors.

Library of Primitive Models: P

$P ::= \langle E_h, (h)E_w, (h,w,x)E_y \rangle$	primitive model

$$\begin{aligned}
\mathsf{CBernoulli} &\triangleq \langle \{\alpha = 1.0; \beta = 1.0\}, \\
&\quad (h)\mathsf{Beta}(h.\alpha, h.\beta), \\
&\quad (h,w,x)\mathsf{Bernoulli}(w) \rangle \\
\mathsf{CGaussian} &\triangleq \langle \{\mu = 0.0; \tau = 1.0; \kappa = 1.0; \theta = 2.0\}, \\
&\quad (h)\{\mu = \mathsf{Gaussian}(h.\mu, h.\tau); \\
&\qquad \tau = \mathsf{Gamma}(h.\kappa, h.\theta)\}, \\
&\quad (h,w,x)\mathsf{Gaussian}(w.\mu, w.\tau) \rangle \\
\mathsf{CDiscrete} &\triangleq \langle \{\mathsf{N} = 2; \alpha = 1.0\}, \\
&\quad (h)\mathsf{DirichletSymmetric}(h.N, h.\alpha), \\
&\quad (h,w,x)\mathsf{Discrete}(w) \rangle
\end{aligned}$$

These models are defined as primitives built from closed Fun expressions. The model CBernoulli is exactly equivalent to our previous example. The concentration α of a CDiscrete determines whether the parameter—a probability vector of length N drawn from the symmetric Dirichlet distribution—is uniformly distributed ($\alpha = 1.0$), biased towards sparse vectors ($\alpha < 1.0$) or dense vectors ($\alpha > 1.0$). Notice that Gaussian is a distribution D that can occur within an expression E, while CGaussian is a primitive model that may occur as a model expression M in the full syntax of Tabular.

Completing Tabular We add primitive and indexed model expressions to enable the succinct expression of complex models.

Completing the Syntax of Model Expressions: M

$M ::=$	model expression
$\quad E$	simple
$\quad P(c_1 = E_1, \ldots, c_n = E_n)$	primitive, with hyperparameters
$\quad M[E_{\text{index}} < E_{\text{size}}]$	indexed

The semantics of a model expression M for a column c is a model P whose output explains how to generate the entry for c in each row of a table. The model P has a restricted form $P = \langle \{\}, (h)E_w, (h,w,x)E_y \rangle$, with no hyperparameters, and where $h \notin \text{fv}(E_w, E_y)$ and $x \notin \text{fv}(E_y)$. Hence, in our notations below, we omit the bound variables h and x.

A simple model E produces its output by running E.

Model for Simple Model Expression E**:**

Hyper	The empty record $\{\}$.
Prior()	The empty record $\{\}$.

Gen(w)	y where $y \sim E$.

A primitive model $P(c = E_c{}^{c \in C'})$ acts like the library model P, except that when P.Hyper $= \{c = F_c{}^{c \in C}\}$ and $C' \subseteq C$, hyperparameter c is set to E_c if $c \in C'$, and otherwise to the default F_c.

Model for $P(c = E_c{}^{c \in C'})$:

Hyper	The empty record $\{\}$.
Prior()	P.Prior($\{c = E_c{}^{c \in C'}; c = F_c{}^{c \in C \setminus C'}\}$).
Gen(w)	P.Gen($\{c = E_c{}^{c \in C'}; c = F_c{}^{c \in C \setminus C'}\}, w, \{\}$).

An indexed model $M[E_{\text{index}} < E_{\text{size}}]$ creates its parameter to be an array of E_{size} instances of the parameter of M, and produces its output like M but using the parameter instance indexed by E_{index}.

Model for $M[E_{\text{index}} < E_{\text{size}}]$ where P is the model for M:

Hyper	The empty record $\{\}$.
Prior()	$[w_1, \ldots, w_{E_{size}}]$ where $w_i \sim P$.Prior() for $i \leq E_{size}$.
Gen(w)	$y \sim P$.Gen(w_i) where $i := E_{\text{index}}$.

Generative Process for Tables in Full Tabular In the full language, the model expression for a column c has both a parameter and an output; we use the variable c\$ for the parameter, and the variable c for the output.

In step (1) the generative process, we process the annotations in turn to define the hyperparameters and parameters.

- $c \mapsto$ **hyper**(E) defines c as the deterministic expression E.
- $c \mapsto$ **param**(M) samples c\$ from P.Prior() and samples c from P.Gen(c\$) where P models M.
- $c \mapsto$ **input** is ignored.
- $c \mapsto$ **output**(M) samples c\$ from P.Prior() where P models M.
- $c \mapsto$ **latent**(M) samples c\$ from P.Prior() where P models M.

In step (2), a loop over each row of the input table, we process the annotations in turn to define each row of the predictive table.

- $c \mapsto$ **hyper**(E) is ignored.
- $c \mapsto$ **param**(M) is ignored.
- $c \mapsto$ **input** copies c from the input row.
- $c \mapsto$ **output**(M) samples c from P.Gen(c\$) where P models M.
- $c \mapsto$ **latent**(M) samples c from P.Gen(c\$) where P models M.

The generative process for the core language is a special case, where the \$ suffixed variables are empty records. As before, the variables defined in step (1) are static variables defined once per table, whereas the variables defined in step (2) are defined for each row of the table. The \$ suffixed variables help define the semantics of Tabular, but are not directly available to Tabular programs.

3.3 Examples of Models and Queries

A mixture model is a probabilistic choice between two or more other models. We begin with several varieties of mixture model.

Mixture of Two Gaussians Our first mixture model makes use of the library models CBernoulli and CGaussian.

MoG1			
z	**bool**	**latent**	CBernoulli()
g1	**real**	**latent**	CGaussian()
g2	**real**	**latent**	CGaussian()
y	**real**	**output**	**if** z **then** g1 **else** g2

In step (1) of the generative process, we sample parameters z\$ (containing the bias) from the prior of CBernoulli(), and parameters g1\$, g2\$ (each containing a mean μ and precision τ) from the prior of CGaussian(). The empty hyperparameter lists in CBernoulli() and CGaussian() indicate that we use the default hyperparameters built into the models, that is, $\{\alpha = 1.0; \beta = 1.0\}$ and $\{\mu = 0.0; \tau = 1.0; \kappa = 1.0; \theta = 2.0\}$.

In step (2), we generate each row of the table by sampling z from the distribution Bernoulli(z\$), g1 and g2 from the distributions Gaussian(g1\$.μ, g1\$.τ) and Gaussian(g2\$.μ, g2\$.τ) and finally defining the output y to be g1 or g2, depending on z.

Given a concrete database for this schema (a column y of random numbers that is expected to be grouped into two clusters around the means of the two Gaussians) inference learns the posterior distributions of the parameters z\$, g1\$, and g2\$, and also fills in the latent columns. The inferred distribution of each z indicates how likely each y is to have been drawn from each of the clusters.

Mixture of an Array of Gaussians To generalize to a many-way mixture, we first decide on a number n of mixture components (clusters); in this case we set n=5. To randomly select a cluster we use the CDiscrete library model, which has an integer hyperparameter N and outputs natural numbers less than N. The default value of N is 2; to define a mixture model with n components we override the default as CDiscrete(N=n). A model CDiscrete(N=2) is akin to a CBernoulli that outputs 0 or 1.

MoG2			
n	**int**	**hyper**	5
z	**int**	**latent**	CDiscrete(N=n)
y	**real**	**output**	CGaussian()[z < n]

The indexed model CGaussian()[z < n] denotes a model whose parameter is an array of n parameter records (containing mean μ and precision τ fields) for the underlying CGaussian model. The output of the indexed model is obtained by first picking the parameter record at index z, and then getting an output from the CGaussian model with those parameters.

The parameter of column z is a *probability vector* of length N, an array of non-negative real numbers that sum to 1, indicating the chance of each output value. The parameter for the y column is an array of n parameter records for the underlying CGaussian model.

The observed output of each row is determined by first sampling the cluster z from the discrete distribution, and then sampling from CGaussian[z < n]. With n=2 we recover our previous mixture of two Gaussians.

User/Movie/Rating Schema Our final mixture model is a Tabular version of the factor graph in Figure 1 of Singh and Graepel (2012), where it was automatically generated from a relational schema.

User			
z	**int**	**latent**	CDiscrete(N=4)
Name	**string**	**input**	
IsMale	**bool**	**output**	CBernoulli()[z]
Age	**int**	**output**	CDiscrete(N=100)[z]
Movie			
z	**int**	**latent**	CDiscrete(N=4)
Title	**string**	**input**	
Genre	**int**	**output**	CDiscrete(N=7)[z]
Year	**int**	**output**	CDiscrete(N=100)[z]
Rating			
u	**link**(User)	**input**	
m	**link**(Movie)	**input**	
Score	**int**	**output**	CDiscrete(N=5)[u.z,m.z]

The model for the Score column illustrates a couple of notations regarding indexed models. First, a doubly-indexed model $M[E_1 < F_1, E_2 < F_2]$ is short for $(M[E_1 < F_1])[E_2 < F_2]$. Second, we write $M[E]$ as short for $M[E < n]$ when we know that E is output by CDiscrete(N=n).

Each row in the User table belongs to one of four clusters, indexed by the latent variable z which has a CDiscrete model.

For each cluster, there is a corresponding distribution over gender (IsMale) and Age. Similarly, each row in the Movie table is modelled by a four-way mixture, indexed by z, with Genre and Year attributes. Finally, each row in the Rating table has links to a user u and to a movie m, and also a Score attribute that is modelled by a discrete distribution indexed by the clusters of the user and the movie, corresponding to a stochastic block model (Nowicki and Snijders 2001).

Bayes Point Machine The Bayes Point Machine (BPM) (Minka 2001) is a Bayesian classification model which takes a vector of floats as input and returns a binary classification. It can be considered the Bayesian equivalent of a linear support vector machine with the added benefits of returning a class probability rather than just a binary decision, and explicitly representing the remaining parameter uncertainty in terms of the posterior distribution over weight vectors. In the Tabular implementation of the BPM the weight vector is endowed with a multivariate Gaussian prior. The inputs X0, X1, and X2 are combined into a feature vector V and the output Y is given by the truth value of a Gaussian distributed score being positive, whose mean is the inner product between the weight vector W and the feature vector V. Inference will provide a Gaussian posterior over the weight vector W which can be used to obtain predictions for test inputs X in the form of Bernoulli distributions over outputs Y.

BPM			
X0	**real**	**input**	
X1	**real**	**input**	
X2	**real**	**input**	
Zero	**vector**	**hyper**	[**for** i<3→0.0]
Unit	**matrix**	**hyper**	[**for** i<3→[**for** j<3→**if** i = j **then** 1.0 **else** 0.0]]
W	**vector**	**param**	VectorGaussian(Zero,Unit)
V	**vector**	**latent**	[X0,X1,X2]
Y	**bool**	**output**	Gaussian(W∗V,1.0)> 0.0

Latent Dirichlet Allocation Latent Dirichlet Allocation (LDA) (Blei et al. 2003) is a powerful yet simple topic model for text, which is widely used to organize text collections and understand the underlying topic structure. Given the building block of a conjugate Discrete model CDiscrete, LDA can be formulated very succinctly within Tabular. The concrete schema has three tables, one for words, one for documents and one for word occurrences. The occurrence table contains input fields Doc and Position which specify a slot for a word. A latent column Topic holds the topic from which the word is being drawn modelled with CDiscrete. Given the topic, the observed column Word is sampled from another conjugate discrete model indexed by Topic. Inference in this model yields distributions over words characterizing each topic as well as distributions over topics for each word occurrence.

Words			
Word	**string**	**input**	
Docs			
Filename	**string**	**input**	
Occs			
Doc	**link**(Docs)	**input**	
Position	**int**	**input**	
NTopics	**int**	**hyper**	10
Topic	**int**	**latent**	CDiscrete(N=NTopics, alpha=15.0)[Doc]
Word	**link**(Words)	**output**	CDiscrete(N=**sizeof**(Words), alpha=0.1)[Topic]

Query-by-Latent-Column and TrueSkill We illustrate direct use of query-by-latent-column with reference to TrueSkill, and also a programming style where we introduce new *query tables* purely for the purpose of formulating queries.

First, as illustrated in Section 1, given tables of players and matches, inference computes distributions for the latent Skill column; these skills can be used to do matchmaking or to display in leaderboards. It also infers distributions for the Perf1 and Perf2 columns, which may indicate whether a player was on form or not on the occasion of a particular match.

Second, suppose we wish to bet on the outcomes of upcoming matches between members p and q of the Players table. We add a fresh query table Bets, which has the same schema as Matches except that Win1 is latent instead of being an observed output. We place one row in this new table, with p for Player1 and q for Player2, and inference computes distributions for the three latent columns, including a Bernoulli for Win1 indicating the odds of a win. By placing multiple rows in the Bets table we can predict the outcomes of multiple upcoming matches.

Bets			
Player1	**link**(Players)	**input**	
Player2	**link**(Players)	**input**	
Perf1	**real**	**latent**	Gaussian(Player1.Skill,1.0)
Perf2	**real**	**latent**	Gaussian(Player2.Skill,1.0)
Win1	**bool**	**latent**	Perf1 > Perf2

Third, consider an online situation where there is a large table of players, and a relatively small number of players q_i queuing to begin fresh online games. We may wish to select one of the q_i to play against a new player p. To do so, we add the Sim query table below, and fill it with rows (p, q_i) for each i. The latent column Similar holds **true** if the two players are close in skill (less than 0.1 units apart). Inference fills this column with Bernoulli distributions which can be used to select a partner close in skill to p. Both the means and variances of the skills of players enter into the marginal probability of being Similar, thus making use of the full probabilistic formulation.

Sim			
Player1	**link**(Players)	**input**	
Player2	**link**(Players)	**input**	
Similar	**bool**	**latent**	abs(Player1.Skill−Player2.Skill)<0.1

4. Formal Semantics of Tabular

4.1 Semantics of Fun (Review)

We here recall the semantics of Fun without zero-probability observations (Bhat et al. 2013). We write $\Gamma \vdash E : T$ to mean that in type environment $\Gamma = x_1 : T_1, \ldots, x_n : T_n$ (x_i distinct) expression E has type T. Let $\mathrm{Det}(E)$ mean that E contains no occurrence of $D(\ldots)$. The typing rules for Fun are standard for a first-order functional language; some examples follow below.

Selected Typing Rules of Fun Expressions: $\Gamma \vdash E : T$

(FUN RANDOM)
$$\frac{D : (x_1 : T_1 * \cdots * x_n : T_n) \to U \quad \Gamma \vdash E_i : T_i \text{ for } i \in 1..n}{\Gamma \vdash D(E_1, \ldots, E_n) : U}$$

(FUN ACONST)
$$\frac{\Gamma \vdash E_i : T \text{ for } i \in 1..n}{\Gamma \vdash [E_1, \ldots, E_n] : T[\,]}$$

(FUN ITER)
$$\frac{\Gamma, x : \textbf{int} \vdash F : T \quad \Gamma \vdash E : \textbf{int} \quad \mathrm{Det}(E)}{\Gamma \vdash [\textbf{for } x < E \to F] : T[\,]}$$

(FUN INDEX)
$$\frac{\Gamma \vdash E : T[\,] \quad \Gamma \vdash F : \textbf{int}}{\Gamma \vdash E[F] : T}$$

The interpretation of a type T is the Borel-measurable set $\mathbf{V}_T$ of closed values of type T (real numbers, integers, records, and so on) using the standard topology. A function $f : T \to U$ is measurable if $f^{-1}(A) \subseteq \mathbf{V}_T$ is measurable for all measurable $A \subseteq \mathbf{V}_U$; all continuous functions are measurable.

A finite measure μ over T is a function from (Borel-measurable) subsets of $\mathbf{V}_T$ to the non-negative real numbers, that is countably additive, that is, $\mu(\cup_i A_i) = \Sigma_i \mu(A_i)$ if $A_1, A_2, \ldots$ are pair-wise disjoint. The finite measure μ is called a probability measure if $\mu(\mathbf{V}_T) = 1.0$. If μ is a probability measure on T and $f : T \to U$

is measurable, we let $f^{-1}\mu(A) \triangleq \mu(f^{-1}(A))$. In this context f is called a *random variable*.

The semantics of a closed Fun expression E is a probability measure P_E over its return type. It is defined via a semantics of open Fun expressions (Ramsey and Pfeffer 2002) in the probability monad (Giry 1982). We write P_E for the probability measure corresponding to a closed expression E; if $\varnothing \vdash E : T$ then P_E is a probability measure on $\mathbf{V}_T$. If $\vdash E : T_1 * \cdots * T_n$, and for $i = 1..m$ we have $\vdash V_i : U_i$ and F_i det and $x_1 : T_1, \ldots, x_n : T_n \vdash F_i : U_i$, we write $\mathrm{P}_E\,[x_1, \ldots, x_n \mid F_1 = V_1 \wedge \cdots \wedge F_m = V_m]$ for (a version of) the conditional probability distribution of P_E given $f = (V_1, \ldots, V_m)$ where $f(x_1, \ldots, x_n) = (F_1, \ldots, F_m)$.

4.2 Semantics of Semi-Observed Models

A model is associated with four types: a hyperparameter type H, a parameter type W, an input type X, and an output type Y.

Model Types and Typing of Models: $Q, \vdash P : Q$

$Q ::= \langle H, W, X, Y \rangle$ quadruple type of model

(TYPE MODEL)

$$\frac{\varnothing \vdash E_h : H \quad \mathrm{Det}(E_h) \quad h : H \vdash E_w : W \quad h : H, w : W, x : X \vdash E_y : Y}{\vdash \langle E_h, (h)E_w, (h,w,x)E_y \rangle : \langle H, W, X, Y \rangle}$$

In a semi-observed model, Y is a pair type, where the second component holds the latent variables of the model. Given a semi-observed model, the standard distributions are obtained as follows.

Proposition 1. *Given a model $P = \langle E_h, (h)E_w, (h,w,x)E_{yz} \rangle$ such that $\vdash P : \langle H, W, X, Y * Z \rangle$ the following Fun expressions denote the standard distributions:*

- *Prior:* **let** $h = E_h$ **in** E_w.
- *Full sampling (where $h = V_h$, $w = V_w$, $x = V_x$):* **let** $h = V_h$ **in let** $w = V_w$ **in let** $x = V_x$ **in** E_{yz}.
- *Sampling (where $h = V_h$, $w = V_w$, $x = V_x$):* **let** $h = V_h$ **in let** $w = V_w$ **in let** $x = V_x$ **in fst** E_{yz}.
- *Joint posterior (where $x = V_X$, $y = V_y$):* $\mathrm{P}_E\,[w, yz \mid \mathbf{fst}\ yz = V_y]$ *where* $E =$ **let** $h = E_h$ **in let** $w = E_w$ **in let** $x = V_x$ **in** w, E_{yz}.
- *Posterior:* $\mathbf{fst}^{-1}\mathrm{P}$ *where* P *is the joint posterior; and*
- *Posterior latent* $(\mathbf{snd} \circ \mathbf{snd})^{-1}\mathrm{P}$ *where* P *is the joint posterior.*

4.3 Typing and Translation of Tabular

One of the purposes of typing Tabular is to catch binding time errors, where identifiers are accidentally defined in terms of other identifiers that are bound later in the computation. Here are some examples of binding time and other errors.

Table1			
c1	int	hyper	3
Bad0	int	hyper	3+c1
c2	bool	latent	Bernoulli(0.8)
Bad1	bool	param	c2
x1	real	input	
Bad2	real	param	x1
x2	int	output	c1
Ok3	int	latent	x2

Bad0 is bad because the default hyperparameter is not closed. Bad1 is bad because it uses a latent variable (defined in step (2)) to define a parameter (defined in step (1)). Bad2 is bad because it uses an input variable (defined in step (2)) to define a parameter. Ok3 is ok because it uses an output (defined in step (2)) to create a latent variable (also defined in step (2)).

When typing schemas, we use *binding times* to track the availability of variables. Let $\mathbf{B}$ be the set $\{\mathbf{h}, \mathbf{w}, \mathbf{xyz}\}$ of binding times ordered such that $\bot = \mathbf{h} < \mathbf{w} < \mathbf{xyz} = \top$. Here $\mathbf{h}$ stands for the (deterministic) hyperparameter phase, $\mathbf{w}$ stands for the (non-deterministic) parameter phase, and $\mathbf{xyz}$ stands for the generative phase of the computation. We use metavariables ℓ and pc to range over $\mathbf{B}$. Informally, variables declared at one time may only be used in expressions typed at or above that time (the current time pc is maintained as an additional index of the Tabular typing judgments). Binding times are also used to prevent the mention of non-deterministic parameters in expressions used as (necessarily deterministic) hyperparameters, and generative data in the construction of either hyperparameters or parameters. When translating to Fun, binding times ensure that the target program is well-scoped, and deterministic where needed. (We considered using a triple of contexts $(\Gamma_{\mathbf{h}}, \Gamma_{\mathbf{w}}, \Gamma_{\mathbf{xyz}})$ instead of annotating each variable binding with a level; overall, it seems syntactically lighter to use binding-time annotations as we have done.)

Tabular Levels and Typing Environments: ℓ, Γ

$\ell, pc ::= \mathbf{h} \mid \mathbf{w} \mid \mathbf{xyz}$	binding time
$\Gamma ::=$	environment
$\varnothing$	empty
$\Gamma, x :^{\ell} T$	variable typing
$\Gamma, t : \langle \{RT\} \rangle$	predictive row type for t

Environments declare variables with their binding time and type, and tables with their predictive row types.

Judgments of the Tabular Type System:

$\Gamma \vdash \diamond$	environment Γ is well-formed
$\Gamma \vdash T$	in Γ, type T is well-formed
$\Gamma \vdash^{pc} E : T$	in Γ at binding time pc, expr. E has type T
$\Gamma \vdash^{pc} M : W, T$	in Γ at pc, model M has params W, returns T
$\Gamma \vdash \mathbb{T} : Q$	in Γ, table $\mathbb{T}$ has type Q
$\Gamma \vdash \mathbb{S} : Q$	in Γ, schema $\mathbb{S}$ has type Q

Formation Rules for Environments: $\Gamma \vdash \diamond$

(ENV EMPTY) $\dfrac{}{\varnothing \vdash \diamond}$ (ENV VAR) $\dfrac{\Gamma \vdash T \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma, x :^{\ell} T \vdash \diamond}$ (ENV TABLE) $\dfrac{\Gamma \vdash \{RT\} \quad t \notin \mathrm{dom}(\Gamma)}{\Gamma, t : \langle \{RT\} \rangle \vdash \diamond}$

Formation Rules for Types: $\Gamma \vdash T$

(TYPE SCALAR) $\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash S}$ (TYPE ARRAY) $\dfrac{\Gamma \vdash T}{\Gamma \vdash T[\,]}$ (TYPE RECORD) $\dfrac{\Gamma \vdash \diamond \quad \forall c \in C.\ \Gamma \vdash T_c}{\Gamma \vdash \{c : T_c\}^{c \in C}}$

The translation of a Tabular schema to a model is performed by four judgments. Though defined relationally, the relations are partial functions on raw terms and total functions on well-typed Tabular terms.

Judgments of the Translation:

$E \Downarrow F$	Tabular expression E translates to Fun expr. F
$M \Downarrow \langle E_w, (w)E \rangle$	model M translates to $\langle E_w, (w)E \rangle$
$\mathbb{T} \Downarrow P$	marked up table $\mathbb{T}$ translates to prim. model P
$\mathbb{S} \Downarrow P$	marked up schema $\mathbb{S}$ translates to P

Lemma 2 (Determinacy)**.** *If $\mathbb{S} \Downarrow P$ and $\mathbb{S} \Downarrow P'$ then $P = P'$.*

Theorem 1 (Translation Preserves Typing)**.**
If $\varnothing \vdash \mathbb{S} : Q$ then there exists P such that $\mathbb{S} \Downarrow P$ and $\vdash P : Q$.

4.4 Expressions

The main subtlety when translating schemas is to support foreign keys. We use the notation $(E : \mathsf{link}(t)).c$ within Fun expressions to stand for the column c of the row in table t indexed by key E.

In particular, when constructing the model for a table t_j, we may dereference a foreign key of type $\mathsf{link}(t_i)$ to a previous table t_i with $i < j$. For instance, in the TrueSkill schema, there is a reference from $t_2 = \mathsf{Matches}$ to $t_1 = \mathsf{Players}$. To translate such foreign keys, we arrange that for each table t_i there is a global variable named t_i that holds the *predictive table* for t_i, that is, the join of the input sub-table x_i, the output sub-table y_i, and the latent sub-table z_i, for each i. Hence, an expression $(E : \mathsf{link}(t_i)).c$ means $t_i[E].c$; for example, $(\mathsf{Player1} : \mathsf{link}(\mathsf{Players}))$.Skill compiles to Players[Player1].Skill.

Typing Rules for Tabular Expressions: $\Gamma \vdash^{pc} E : T$

(TABULAR VAR)
$$\frac{\Gamma \vdash \diamond \quad \Gamma = \Gamma_1, x :^{\ell} T, \Gamma_2 \quad \ell \leq pc}{\Gamma \vdash^{pc} x : T}$$

(TABULAR CONST)
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash^{pc} s : \mathrm{ty}(s)}$$

(TABULAR PRIM)
$$\frac{\Gamma \vdash \diamond \quad g : (x_1 : T_1, \ldots, x_n : T_n) \to T \quad \Gamma \vdash^{pc} E_i : T_i \quad \forall i \in 1..n}{\Gamma \vdash^{pc} g(E_1, \ldots, E_n) : T}$$

(TABULAR RANDOM)
$$\frac{\Gamma \vdash \diamond \quad D : (x_1 : T_1, \ldots, x_n : T_n) \to T \quad \Gamma \vdash^{pc} E_i : T_i \quad \forall i \in 1..n}{\Gamma \vdash^{pc} D(E_1, \ldots, E_n) : T}$$

(TABULAR IF)
$$\frac{\Gamma \vdash^{pc} E_1 : \mathsf{bool} \quad \Gamma \vdash^{pc} E_2 : T \quad \Gamma \vdash^{pc} E_3 : T}{\Gamma \vdash^{pc} \mathsf{if}\ E_1\ \mathsf{then}\ E_2\ \mathsf{else}\ E_3 : T}$$

(TABULAR ARRAY)
$$\frac{\Gamma \vdash \diamond \quad \Gamma \vdash^{pc} E_i : T \quad \forall i \in 1..n}{\Gamma \vdash^{pc} [E_1, \ldots, E_n] : T[]}$$

(TABULAR ITER)
$$\frac{\Gamma \vdash^{pc} E : \mathsf{int} \qquad \Gamma, x :^{pc} \mathsf{int} \vdash^{pc} F : T}{\Gamma \vdash^{pc} [\mathsf{for}\ x < E \to F] : T[]}$$

(TABULAR INDEX)
$$\frac{\Gamma \vdash^{pc} E : T[] \qquad \Gamma \vdash^{pc} F : \mathsf{int}}{\Gamma \vdash^{pc} E[F] : T}$$

(TABULAR LET)
$$\frac{\Gamma \vdash^{pc} E_1 : T_1 \quad \Gamma, x :^{pc} T_1 \vdash^{pc} E_2 : T_2}{\Gamma \vdash^{pc} \mathsf{let}\ x = E_1\ \mathsf{in}\ E_2 : T_2}$$

(TABULAR SIZEOF)
$$\frac{\Gamma \vdash^{pc} \#t : \mathsf{int} \quad t \in \mathrm{dom}(\Gamma)}{\Gamma \vdash^{pc} \mathsf{sizeof}(t) : \mathsf{int}}$$

(TABULAR DEREF)
$$\frac{\Gamma \vdash^{pc} E : \mathsf{int} \quad \mathbf{xyz} \leq pc \quad \Gamma = \Gamma', t : \langle \{d : T_d\}^{d \in C} \rangle, \Gamma'' \quad c \in C}{\Gamma \vdash^{pc} (E : \mathsf{link}(t)).c : T_c}$$

Rule (TABULAR VAR) allows a reference to x only if x is declared with a binding time $l \leq pc$, where pc is the current binding time.

Translation Rules for Tabular Expressions: $E \Downarrow F$

(TRANS VAR)
$$\frac{}{x \Downarrow x}$$

(TRANS CONST)
$$\frac{}{s \Downarrow s}$$

(TRANS PRIM)
$$\frac{E_i \Downarrow F_i \quad \forall i \in 1..n}{g(E_1, \ldots, E_n) \Downarrow g(F_1, \ldots, F_n)}$$

(TRANS RANDOM)
$$\frac{E_i \Downarrow F_i \quad \forall i \in 1..n}{D(E_1, \ldots, E_n) \Downarrow D(F_1, \ldots, F_n)}$$

(TRANS IF)
$$\frac{E_i \Downarrow F_i \quad \forall i \in 1..3}{\mathsf{if}\ E_1\ \mathsf{then}\ E_2\ \mathsf{else}\ E_3 \Downarrow \mathsf{if}\ F_1\ \mathsf{then}\ F_2\ \mathsf{else}\ F_3}$$

(TRANS ARRAY)
$$\frac{E_i \Downarrow F_i \quad \forall i \in 1..n}{[E_1, \ldots, E_n] \Downarrow [F_1, \ldots, F_n]}$$

(TRANS ITER)
$$\frac{E_i \Downarrow F_i \quad \forall i \in 1..2}{[\mathsf{for}\ x < E_1 \to E_2] \Downarrow [\mathsf{for}\ x < F_1 \to F_2]}$$

(TRANS INDEX)
$$\frac{E_i \Downarrow F_i \quad \forall i \in 1..2}{E_1[E_2] \Downarrow F_1[F_2]}$$

(TRANS LET)
$$\frac{E_i \Downarrow F_i \quad \forall i \in 1..2}{\mathsf{let}\ x = E_1\ \mathsf{in}\ E_2 \Downarrow \mathsf{let}\ x = F_1\ \mathsf{in}\ F_2}$$

(TRANS SIZEOF)
$$\frac{}{\mathsf{sizeof}(t) \Downarrow \#t}$$

(TRANS DEREF)
$$\frac{E \Downarrow F}{(E : \mathsf{link}(t)).c \Downarrow t[F].c}$$

4.5 Model Expressions

Typing Rules for Model Expressions: $\Gamma \vdash^{pc} M : W, T$

(MODEL SIMPLE)
$$\frac{\Gamma \vdash^{pc} E : T}{\Gamma \vdash^{pc} E : \{\}, T}$$

(MODEL PRIM)
$$\frac{\begin{array}{l}\Gamma \vdash \diamond \quad P = \langle \{R\}, (h)E_w, (h,w,x)E_y \rangle \\ \vdash P : \langle \{c : H_c\}^{c \in C}, W, \{\}, Y \rangle \\ \forall c \in C' \subseteq C. \quad \Gamma \vdash^{\mathbf{h}} E_c : H_c \ \wedge\ \mathrm{Det}(E_c)\end{array}}{\Gamma \vdash^{pc} P(c = E_c{}^{c \in C'}) : W, Y}$$

(MODEL INDEXED)
$$\frac{\Gamma \vdash^{pc} M : W, T \quad \Gamma \vdash^{pc} E_{\text{index}} : \mathsf{int} \quad \Gamma \vdash^{\mathbf{h}} E_{\text{size}} : \mathsf{int} \quad \mathrm{Det}(E_{\text{size}})}{\Gamma \vdash^{pc} M[E_{\text{index}} < E_{\text{size}}] : W[], T}$$

Primitive models must have void input; we allow to only replace a part C' of their hyperparameters C. The upper bound E_{size} of an indexed model has binding time $\mathbf{h}$, since it must be deterministic and the same for all rows of the table.

Translation Rules for Model Expressions: $M \Downarrow P$

(TRANS SIMPLE) $(w \notin \mathrm{fv}(F))$
$$\frac{E \Downarrow F}{E \Downarrow \langle \{\}, (w)F \rangle}$$

(TRANS MODEL PRIM) $(w \notin \mathrm{fv}(E_h))$
$$\frac{\begin{array}{l}P = \langle \{c = F_c\}^{c \in C}, (h)E_w, (h,w,x)E_y \rangle \qquad E_c \Downarrow E'_c\ (c \in C') \\ \hat{E}_c = \text{if } c \in C' \text{ then } E'_c \text{ else } F_c \qquad E_h = \{c = \hat{E}_c\}^{c \in C}\end{array}}{\begin{array}{l}P(c = E_c{}^{c \in C'}) \Downarrow \\ \langle \mathsf{let}\ h = E_h\ \mathsf{in}\ E_w, (w)\mathsf{let}\ h = E_h\ \mathsf{in}\ \mathsf{let}\ x = \{\}\ \mathsf{in}\ E_y \rangle\end{array}}$$

(TRANS INDEXED) $(w \notin \mathrm{fv}(F_{\text{index}}))$
$$\frac{E_{\text{index}} \Downarrow F_{\text{index}} \qquad E_{\text{size}} \Downarrow F_{\text{size}} \qquad M \Downarrow \langle E_w, (w)E_y \rangle}{\begin{array}{l}M[E_{\text{index}} < E_{\text{size}}] \Downarrow \\ \langle [\mathsf{for}\ \_ < F_{\text{size}} \to E_w], (w)\mathsf{let}\ w = w[F_{\text{index}}]\ \mathsf{in}\ E_y \rangle\end{array}}$$

A simple model has no prior. The prior of an indexed model is an array of F_{size} independent samples of the prior of the underlying model. In the output, we use the prior value at index F_{index}.

4.6 Tables

The typing and translation rules for tables are defined inductively and determine the semantics for the shared hyperparameter, shared

parameter, and a pair of output and latent columns for a single row of the table.

Typing Rules for Tables: $\Gamma \vdash \mathbb{T} : Q$

(TABLE EMPTY)
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \varnothing : \langle \{\}, \{\}, \{\}, \{\} * \{\} \rangle}$$

(TABLE HYPER) ($A = \textbf{hyper}(E)$)
$$\frac{\varnothing \vdash^{\mathbf{h}} E : H \quad \mathrm{Det}(E) \quad \Gamma, c :^{\mathbf{h}} H \vdash \mathbb{T} : \langle \{RH\}, W, X, Y * Z \rangle}{\Gamma \vdash (c \mapsto A : H)\mathbb{T} : \langle \{c : H; RH\}, W, X, Y * Z \rangle}$$

(TABLE PARAM) ($A = \textbf{param}(M)$)
$$\frac{\Gamma \vdash^{\mathbf{w}} M : W_{\$}, W \quad \Gamma, c :^{\mathbf{w}} W \vdash \mathbb{T} : \langle H, \{RW\}, X, Y * Z \rangle \quad c\$ \notin \mathrm{dom}(\Gamma) \cup \mathrm{dom}(\mathbb{T})}{\Gamma \vdash (c \mapsto A : W)\mathbb{T} : \langle H, \{c\$: W_{\$}; c : W; RW\}, X, Y * Z \rangle}$$

(TABLE INPUT) ($A = \textbf{input}$)
$$\frac{\Gamma, c :^{\mathbf{xyz}} X \vdash \mathbb{T} : \langle H, W, \{RX\}, Y * Z \rangle}{\Gamma \vdash (c \mapsto A : X)\mathbb{T} : \langle H, W, \{c : X; RX\}, Y * Z \rangle}$$

(TABLE OUTPUT) ($A = \textbf{output}(M)$)
$$\frac{\Gamma \vdash^{\mathbf{xyz}} M : W, Y \quad \Gamma, c :^{\mathbf{xyz}} Y \vdash \mathbb{T} : \langle H, \{RW\}, X, \{RY\} * Z \rangle}{\Gamma \vdash (c \mapsto A : Y)\mathbb{T} : \langle H, \{c\$: W; RW\}, X, \{c : Y; RY\} * Z \rangle}$$

(TABLE LATENT) ($A = \textbf{latent}(M)$)
$$\frac{\Gamma \vdash^{\mathbf{xyz}} M : W, Z \quad \Gamma, c :^{\mathbf{xyz}} Z \vdash \mathbb{T} : \langle H, \{RW\}, X, Y * \{RZ\} \rangle}{\Gamma \vdash (c \mapsto A : Z)\mathbb{T} : \langle H, \{c\$: W; RW\}, X, Y * \{c : Z; RZ\} \rangle}$$

Rule (TABLE HYPER) ensures that E is deterministic and closed and declares c at binding time **h** so it can be referenced at all binding times. Rule (TABLE PARAM) ensures that M is checked at level **w** (not *pc*) so that its generative expression has no data dependencies and is safe to use at the parameter level. Rule (TABLE INPUT) extends the context with c declared at **xyz**. Rule (TABLE OUTPUT) extends the context with c declared at **xyz** and records the types of parameter $c\$$ and output c by extending the parameter and output record types of the table. Rule (TABLE LATENT) is symmetric to (TABLE OUTPUT), but instead extends the latent record type.

The translation rules for tables make use of auxiliary *let-contexts*, ranged over by $\mathscr{L}$. These denote a spine of (Fun) **let**-bindings ending in a hole $[]$, and are defined inductively as follows.

(Core Fun) Let contexts: $\mathscr{L}$

$\mathscr{L} ::=$	let context
$[]$	hole
$\textbf{let}\ x = E\ \textbf{in}\ \mathscr{L}$	let binding

The operation $\mathscr{L}[E]$ plugs the hole of a $\mathscr{L}$ with a body E, producing a (Fun) expression.

$$\begin{aligned} [][E] &= E \\ (\textbf{let}\ x = E'\ \textbf{in}\ \mathscr{L})[E] &= \textbf{let}\ x = E'\ \textbf{in}\ (\mathscr{L}[E]) \end{aligned}$$

Translation Rules for Tables: $\mathbb{T} \Downarrow P$

(TRANS EMPTY TABLE)
$$\frac{}{\varnothing \Downarrow \langle \{\}, (h)\{\}, (h,w,x)(\{\}, \{\}) \rangle}$$

(TRANS HYPER) ($c \notin \{h,w,x\}$)
$$\frac{E \Downarrow E_h \quad \mathbb{T} \Downarrow \langle \{R_h\}, (h)E_w, (h,w,x)E \rangle}{\begin{array}{l}(c \mapsto \textbf{hyper}\ E : T_c)\mathbb{T} \Downarrow \\ \quad \langle \{c = E_h, R_h\}, (h)\textbf{let}\ c = h.c\ \textbf{in}\ E_w, (h,w,x)\textbf{let}\ c = h.c\ \textbf{in}\ E \rangle\end{array}}$$

(TRANS PARAM) ($h \notin \mathrm{fv}(E_w, E_c, c\$, c) \quad c \notin \{h,w,x\}$)
$$\frac{M \Downarrow \langle E_w, (w_c)E_c \rangle \quad \mathbb{T} \Downarrow \langle E_h, (h)\mathscr{L}_w[\{R_w\}], (h,w,x)E \rangle}{\begin{array}{l}(c \mapsto \textbf{param}\ M : T_c)\mathbb{T} \Downarrow \\ \quad \langle E_h,\ (h)\textbf{let}\ c\$ = E_w\ \textbf{in let}\ c = E_c\ \textbf{in}\ \mathscr{L}_w[\{c\$ = c\$; c = c; R_w\}], \\ \quad (h,w,x)\textbf{let}\ c = w.c\ \textbf{in}\ E \rangle\end{array}}$$

(TRANS INPUT)
$$\frac{\mathbb{T} \Downarrow \langle E_h, (h)E_w, (h,w,x)E \rangle \quad c \notin \{h,w,x\}}{(c \mapsto \textbf{input} : T_c)\mathbb{T} \Downarrow \langle E_h, (h)E_w, (h,w,x)\textbf{let}\ c = x.c\ \textbf{in}\ E \rangle}$$

(TRANS OUTPUT) ($h \notin \mathrm{fv}(E_w) \quad h,w,x \notin \mathrm{fv}(E_c, c)$))
$$\frac{M \Downarrow \langle E_w, (w_c)E_c \rangle \quad \mathbb{T} \Downarrow \langle E_h, (h)\mathscr{L}_w[\{R_w\}], (h,w,x)\mathscr{L}_o[(\{R_y\}, E_z)] \rangle}{\begin{array}{l}(c \mapsto \textbf{output}\ M : T_c)\mathbb{T} \Downarrow \\ \quad \langle E_h,\ (h)\textbf{let}\ c = E_w\ \textbf{in}\ \mathscr{L}_w[\{c\$ = c, R_w\}], \\ \quad (h,w,x)\textbf{let}\ c = (\textbf{let}\ w_c = w.c\$\ \textbf{in}\ E_c)\ \textbf{in}\ \mathscr{L}_o[(\{c = c; R_y\}, E_z)] \rangle\end{array}}$$

(TRANS LATENT) ($h \notin \mathrm{fv}(E_w) \quad h,w,x \notin \mathrm{fv}(E_c, c)$)
$$\frac{M \Downarrow \langle E_w, (w_c)E_c \rangle \quad \mathbb{T} \Downarrow \langle E_h, (h)\mathscr{L}_w[\{R_w\}], (h,w,x)\mathscr{L}_o[(E_y, \{R_z\})] \rangle}{\begin{array}{l}(c \mapsto \textbf{latent}\ M : T_c)\mathbb{T} \Downarrow \\ \quad \langle E_h,\ (h)\textbf{let}\ c = E_w\ \textbf{in}\ \mathscr{L}_w[\{c\$ = c, R_w\}], \\ \quad (h,w,x)\textbf{let}\ c = (\textbf{let}\ w_c = w.c\$\ \textbf{in}\ E_c)\ \textbf{in}\ \mathscr{L}_o[(E_y, \{c = c; R_z\})] \rangle\end{array}}$$

Rule (TRANS HYPER) merely extends the hyperparameter record of the remaining table and rebinds c as the projection $h.t$ in the prior and gen of the model. Rule (TRANS PARAM) extends table $\mathbb{T}$'s prior with two fields for the prior and gen of M, and rebinds parameter c as the projection $w.c$ in the gen of the row. Rule (TRANS INPUT) just binds c as the projection $x.c$ of input row x in the gen of the table (but does not export c since it is neither output nor latent). Rule (TRANS OUTPUT) just defines c as the gen of its model, whose parameter w_c is obtained from $w.c\$$; c is exported in the output record of the row. Rule (TRANS LATENT) is symmetric to (TRANS OUTPUT), but instead extends the latent record.

For example, here is a single-table schema for linear regression.

LinearRegression			
muA	**real**	**hyper**	0
muB	**real**	**hyper**	0
A	**real**	**param**	Gaussian(muA,1)
B	**real**	**param**	Gaussian(muB,1)
X	**real**	**input**	
Z	**real**	**latent**	A∗X + B
Y	**real**	**output**	Gaussian(Z,1)

The row semantics of this table is as follows. For readability, we inline some variable definitions. Since this table only uses simple model expressions, the $ suffixed fields for the parameters of model expressions all contain the empty record. Modulo these redundant fields, we recover the model from Section 2.

Model for a Row of the LinearRegression **Table:**

Hyper	$\{\mathsf{muA} = 0; \mathsf{muB} = 0\}$
Prior(h)	$\{\mathsf{A\$} = \{\}; \mathsf{A} = \mathsf{Gaussian}(h.\mathsf{muA}, 1);$
	$\mathsf{A\$} = \{\}; \mathsf{B} = \mathsf{Gaussian}(h.\mathsf{muB}, 1);$
	$\mathsf{Z\$} = \{\}; \mathsf{Y\$} = \{\}\}$
Gen(h,w,x)	**let** Z = w.A* x.X + w.B **in**
	let Y = Gaussian(Z,1)} **in**
	({Y=Y},{Z=Z})

4.7 Schemas

The typing and translation rules are defined inductively.

Typing Rules for Schemas: $\Gamma \vdash \mathbb{S} : Q$

(SCHEMA EMPTY)
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \varnothing : \langle \{\}, \{\}, \{\}, \{\} * \{\} \rangle}$$

(SCHEMA TABLE)

$$\frac{\begin{array}{l}\Gamma \vdash \mathbb{T} : \langle H, W, \{RX_t\}, \{RY_t\} * \{RZ_t\}\rangle \\ \Gamma, \#t :^{\mathbf{h}} \mathbf{int}, t : \langle \{RX_t; RY_t; RZ_t\}\rangle \vdash \\ \quad \mathbb{S} : \langle \{RH\}, \{RW\}, \{RX\}, \{RY\} * \{RZ\}\rangle \\ H' = \{\#t : \mathbf{int}; RH\} \quad W' = \{t : W; RW\} \quad X' = \{t : \{RX_t\}[]; RX\} \\ Y' = \{t : \{RY_t\}[]; RY\} \quad Z' = \{t : \{RZ_t\}[]; RZ\}\end{array}}{\Gamma \vdash (t \mapsto \mathbb{T})\mathbb{S} : \langle H', W', X', Y' * Z'\rangle}$$

Rule (SCHEMA TABLE) uses the model type of the table to extend the context with a declaration of the table's size, $\#t$ at level $\mathbf{h}$, ($\#t$ is used in the translation of $\mathbf{sizeof}(t)$) as well as the predictive row type of t: this is the union of its input, output, and latent fields. The table's default hyperparameters (of type H) are applied in the translation of t and do not appear in the type of the schema. The rule extends the components of the schema's model type with additional fields for the table size; the parameters of the table (as a nested record); the inputs of the table (a nested *array* of records); and the pair of output and latent table records extended with fields for the output and latent *arrays* of records for t.

Translation Rules for Schemas: $\mathbb{S} \Downarrow P$

(TRANS EMPTY SCHEMA)

$$\varnothing \Downarrow (\{\}, (h)\{\}, (h,w,x)\{\})$$

(TRANS TABLE)

$$\frac{\begin{array}{l}\mathbb{T} \Downarrow \langle E_h, (h_t)E_w, (h_t, w_t, x_i)\mathscr{L}_t[\{R_y\}, \{R_z\}]\rangle \\ R_x = \{c = x_i.c \mid c \in \mathsf{inputs}(\mathbb{T})\} \\ \mathbb{S} \Downarrow (\{R_h\}, (h)\mathscr{L}_w[\{R_w\}], (h,w,x)\mathscr{L}_{yz}[\{S_y\}, \{S_z\}] \\ E_t = \mathbf{let}\ h_t = E_h\ \mathbf{in}\ \mathbf{let}\ w_t = w.t\ \mathbf{in} \\ \quad [\mathbf{for}\ i < \#t \to \mathbf{let}\ x_i = x.t[i]\ \mathbf{in}\ \mathscr{L}_t[\{R_x; R_y; R_z\}]] \\ E_y = [\mathbf{for}\ i < \#t \to \{c = t[i].c\}^{c \in \mathrm{dom}(R_y)}] \\ E_z = [\mathbf{for}\ i < \#t \to \{c = t[i].c\}^{c \in \mathrm{dom}(R_z)}] \\ h \notin \mathrm{fv}(\mathbf{let}\ h_t = E_h\ \mathbf{in}\ E_w, t, \#t) \quad h, w, x \notin \mathrm{fv}(E_t, t, \#t)\end{array}}{\begin{array}{l}(t \mapsto \mathbb{T})\mathbb{S} \Downarrow \\ \langle \{\#t = 1, R_h\}, \\ (h)\mathbf{let}\ t = \mathbf{let}\ h_t = E_h\ \mathbf{in}\ E_w\ \mathbf{in}\ \mathbf{let}\ \#t = h.\#t\ \mathbf{in}\ \mathscr{L}_w[\{t = t; R_w\}], \\ (h,w,x)\mathbf{let}\ \#t = h.\#t\ \mathbf{in}\ \mathbf{let}\ t = E_t\ \mathbf{in} \\ \quad \mathscr{L}_{yz}[(\{t = E_y; S_y\}, \{t = E_z; S_z\})]\rangle\end{array}}$$

Rule (TRANS TABLE) takes the model for the parameters and a single row of t and constructs a model that draws once from the prior of t then replicates t's output distribution across an array of size $\#t$. The intermediate array, E_t, contains the predictive table for t, merging the input, output and latent sub-records of t as single records. Expressions E_y and E_z are used to reshuffle the array of merged records into separate arrays of output and latent sub-records. The rule extends $\mathbb{S}$'s hyperparameter record with a default binding for $\#t$ (with arbitrary value 1); table sizes must be consistently overriden before inference.

4.8 Translation examples

To illustrate our schema translation and our treatment of foreign keys, here is the translation of TrueSkill, rewritten a little for readability: first, the two row models for the two tables, followed by the model of the whole schema.

Model for a Row of Table Players**:** P_1

```
Hyper        {}
Prior(h)     {Skill$ = {}}
Gen(h,w,x)   let Skill =Gaussian(25,0.01) in
             ({}, {Skill = Skill})
```

Model for a Row of Table Matches**:** P_2

```
Hyper        {}
Prior(h)     {Perf1$ = {}; Perf2$ = {}; Win1$ = {}}
Gen(h,w,x)   let Perf1=Gaussian(Players[x.Player1].Skill,1) in
             let Perf2=Gaussian(Players[x.Player2].Skill,1) in
             let Win1= Perf1 > Perf2 in
             ({Win1 = Win1}, {Perf1 = Perf1; Perf2 = Perf2})
```

Model for the TrueSkill Schema:

```
Hyper        {#Players = 1, #Matches = 1}
Prior(h)     {Players = P1.Prior(P1.Hyper),
               Matches = P2.Prior(P2.Hyper)}
Gen(h,w,x)
let Players = [for i <h.#Players →
    let Skill = Gaussian(25,0.01) in
    {Skill = Skill}]
let Matches = [for i <h.#Matches →
    let Player1 = x.Matches[i].Player1 in
    let Player2 = x.Matches[i].Player2 in
    let Perf1=Gaussian(Players[Player1].Skill,1) in
    let Perf2=Gaussian(Players[Player2].Skill,1) in
    let Win1= Perf1 > Perf2 in
    ({Player1 = Player1; Player2 = Player2;
      Win1 = Win1; Perf1 = Perf1; Perf2 = Perf2}) ]
({  Players = [for i < h.#Players → {}];
    Matches = [for i < h.#Matches →
              {Win1 = Matches[i].Win1}]},
 {  Players = [for i < h.#Players → {Skill = Players[i].Skill}];
    Matches = [for i < h.#Matches →
    {Perf1 = Matches[i].Perf1; Perf2 = Matches[i].Perf2}}])
```

4.9 A Reference Learner for Query-by-Latent-Column

We conclude with a *learner API*, a programming interface for query-by-latent-column: the API allows a user to accumulate a dataset split into input and observed databases. To perform queries, we bundle a database and a schema into a *learner* $L = (d \mid \mathbb{S})$ where $d = (d_x, d_y)$ and d_x is the input database and d_y is the observed database. (We assume the types of d and $\mathbb{S}$ match, as discussed in the next section.) To pick out the sizes of tables in a database, we let $\#(\{t_1 = B_1; \ldots; t_n = B_n\}) \triangleq \{\#t_1 = |B_1|; \ldots; \#t_n = |B_n|\})$. We support the following functional API.

- Let $L_0(\mathbb{S})$ be the *empty learner*, that is, $\mathbb{S}$ plus a pair of databases with the right table names but no table rows.
- Let $\mathsf{train}(L, (d'_x, d'_y))$ be $L' = ((d_x + d'_x, d_y + d'_y) \mid \mathbb{S})$ where $+$ is concatenation of arrays in records, and $L = ((d_x, d_y) \mid \mathbb{S})$.
- Let $\mathsf{params}(L)$ be the posterior distribution $p(w \mid d, h)$ induced by P, where $L = (d \mid \mathbb{S})$, P models $\mathbb{S}$, and $h = \#(d_x)$.
- Let $\mathsf{latents}(L)$ be the posterior latent distribution $p(z \mid d, h)$ induced by P, where $L = (d \mid \mathbb{S})$, P models $\mathbb{S}$, and $h = \#(d_x)$.

Compared to the reference learner of Gordon et al. (2013), this new API can learn latent outputs since it works on semi-observed models. Our current implementation uses Infer.NET Fun to compute approximate marginal forms of the posterior distributions on the database parameter and latent database, and persists them to the relational store. The API allows an incremental implementation, where the abstract state L is represented by a distribution over the parameters and latent variables, computed after each call to train. Our current implementation does not support this optimization, maintains the whole dataset d, and does inference from scratch when necessary. The incremental formulation of our learner is consistent with the Algebraic Classifier formulation of Izbicki

(2013), which promises reductions in computational complexity for cross-validation and enable efficient online and parallel training algorithms based on the monoidal or group structure of such learners.

Now that we have schema typing and a semantics of schemas as models, we can perform inference as follows. Let a learner $L = (d_x, d_y \mid \mathbb{S})$ be *queryable* if $\vdash \mathbb{S} : \langle H, W, X, Y * Z \rangle$ and $\varnothing \vdash d_x : X$ and $\varnothing \vdash d_y : Y$, and for all tables $t_i \in \mathrm{dom}(\mathbb{S})$ we have $|d_x.t_i| = |d_y.t_i| \geq 1$. In particular, the empty learner is not queryable, since it contains empty tables. We can now implement a latent column query.

Theorem 2. *If $L = (d_x, d_y \mid \mathbb{S})$ is queryable, there is a closed Fun expression $E(d_x)$ such that if $\mu \triangleq \mathrm{P}_{E(d_x)}\left[w, yz \mid \mathsf{fst}\ yz = d_y\right]$ then*

(1) $\mathsf{params}(L) = \mathsf{fst}^{-1}\mu$*; and*
(2) $\mathsf{latents}(L) = (\mathsf{snd} \circ \mathsf{snd})^{-1}\mu$.

Proof: Assume that $\mathbb{S} \Downarrow \langle E_h, (h)E_w, (h,w,x)E_{yz} \rangle$, and let expression $E(d_x) \triangleq \mathsf{let}\ h = \#(d_x)\ \mathsf{in}\ \mathsf{let}\ w = E_w\ \mathsf{in}\ \mathsf{let}\ x = d_x\ \mathsf{in}\ w, E_{yz}$. By Proposition 1, μ as above yields the sought distributions. ■

5. Outline of Practical Implementation

Our implementation builds on the model-learner pattern of Gordon et al. (2013), in which models are represented as records of type-indexed F# quotations representing typed Fun expressions. Our initial Tabular implementation generates such strongly-typed models. This target confers two advantages: the quotation fragments are compact yet statically checked for type correctness; the resulting terms are easily JIT-compiled to produce efficient sampling code. The latter may be used to generate sampled outputs from user-provided inputs (which may be synthetic or real data) and is a useful tool for testing models.

For clarity, the semantics in Section 4 splits compilation into type-checking followed by untyped translation. To create strongly-typed quotations, we need to convince F#'s type checker that our dynamically constructed quotations are composed in a statically safe manner. The most direct way to do so is to re-structure the separate typing and translation judgments as single elaboration judgments that couple type-checking with translation. The F# rendition of this idea is a triple of polymorphic functions that represent the typing contexts as a pair of (nested) tuples. Contexts are extended as required by using polymorphic recursion in recursive calls to elaboration. The output of elaboration is a value of existential type containing both the target type and the target translation of the source term. Since type variables have accurate run-time representations in .NET, we can directly compare the types of generated sub-expressions as needed, avoiding the need to maintain separate type representations.

While an interesting implementation technique, it also has some drawbacks — F# records are nominal, not structural, so difficult to quote dynamically (in the absence of appropriate record type declarations). Our implementation must normalise Tabular records to nested pairs, similar to the way we encode contexts. Expressing the translation using static quotations also precludes the preservation of schema-specific variable names, which can make the generated code challenging to decipher when debugging the compiler. Finally, our use of existential types, which are not directly supported in F#, requires an awkward encoding via generic classes.

A secondary role of the elaborator is to construct schema derived functions for reading and writing concrete data and distributions to the database using a library. Unfortunately, since the schema of the database is not statically known, the language integrated query facilities of F# are of no direct help in implementing this functionality.

Figure 1 reports example compile and inference times for some of the models described here. Column *S/R* indicates the use of synthetic or real data; *Table Sizes* gives the size of the tables (number of records per table), *T2F* is the Tabular to Fun compilation time, *F2IN* is the Fun to Infer.NET compilation time and *IN* is Infer.NET's internal compilation time (all in milliseconds). While translating Tabular to Fun is cheap, compilation times are dominated by the *F2IN* phase, due to a performance problem with the Fun compiler.

All performance numbers reported here are based on Infer.NET's Expectation Propatation algorithm.

6. Case Study: Intelligence Testing

Tabular has been designed to make the paradigm of model-based machine learning (Bishop 2013) usable for practitioners who are not machine learning experts. We describe a case study of data analysis using Tabular based on a dataset from intelligence testing.

Our case study relies on models first published by Bachrach et al. (2012) and data provided by the Cambridge Psychometrics Centre, based on testing material by Pearson Assessment. We use a dataset of responses to a standard multiple-choice intelligence test called Raven's Standard Progressive Matrices (SPM). The test consists of sixty questions, each comprising a matrix of shapes with one element missing and eight possible answers, exactly one of which is correct. The sample consists of 121 subjects who filled SPM for its standardization in the British market in 2006. The factor graph for the full Difficulty-Ability-Response (DARE) model is shown in Figure 2. Responses and true answers may or may not be observed.

Figure 2 also depicts the full DARE model in Tabular. Each participant is characterized by a latent Ability. Each question is characterized by a (true) Answer, a Difficulty and a Discrimination parameter. Responses depend on ParticipantID and QuestionID. Under the model, an Advantage variable is calculated as the difference between ability of participant and difficulty of question. The Boolean variable Know, which represents whether the participant knows the answer or not, is modelled as a probit over Advantage with Discrimination as the dispersion parameter. DB returns its first argument, and is a pragma to the underlying inference algorithm, to apply a damping factor for better convergence. The primitive operator Probit(a,b) is equivalent to Gaussian(a,b)>0.0). Guess represents a random guess from a uniform distribution over all possible responses. The participant's Response is taken to be the question Answer if Know is true and Guess otherwise. The model relies on two sources of observed data: correct answers to the questions and responses provided by students. A subset of correct answers can be provided through the table QuestionsTrain. A subset of given responses can be provided through the table ResponsesTrain.

Note that there are simplified versions of the full DARE model in which a) only the student's ability is modelled (A model) or b) the students' abilities and the questions' difficulties are modelled (DA model). The model is run once to answer two types of queries given a subset of the true answers and a subset of given responses: i) Infer the missing correct answers to questions and ii) Infer the missing responses of students.

Figure 3 shows how the Tabular implementation differs from the Infer.NET implementation on a sample run where 30% of responses and 30% of true answers are unobserved. The data contained 121 participants, 60 questions, 41 training questions, 7260 responses and 5082 training responses. The inference results of Infer.NET and Tabular based implementations are very similar. They differ slightly because of differences in the way our compiler translated the Tabular formulation into Infer.NET code from the direct implementation by an expert. However, the Infer.NET code includ-

Model	S/R	Table Sizes	T2F(ms)	F2IN(ms)	IN	Inference(ms)
TrueSkill	S	100,20000	621	54345	1286	49737
Recommender	S	20,200,100	706	11241	3609	637
RecommenderQuery	S	20,200,100,20	775	33230	3916	26333
InfernoClassicMM	S	100,33,33,33	512	14003	3385	793

Figure 1. Tabular Benchmarks

Participants			
Ability	**real**	**latent**	Gaussian(0.0,1.0)
Questions			
Answer	**int**	**latent**	DiscreteUniform(8)
Difficulty	**real**	**latent**	Gaussian(0.0,1.0)
Discrimination	**real**	**latent**	Gamma(5.0,0.2)
QuestionsTrain			
QuestionID	**link**(Questions)	**input**	
Answer	**int**	**output**	QuestionID.Answer
Responses			
ParticipantID	**link**(Participants)	**input**	
QuestionID	**link**(Questions)	**input**	
Advantage	**real**	**latent**	DB(ParticipantID.Ability − QuestionID.Difficulty,0.2)
Know	**bool**	**latent**	Probit(Advantage,QuestionID.Discrimination)
Guess	**int**	**latent**	DiscreteUniform(8)
Response	**int**	**latent**	**if** Know **then** QuestionID.Answer **else** Guess
ResponsesTrain			
ResponseID	**link**(Responses)	**input**	
Response	**int**	**output**	ResponseID.Response

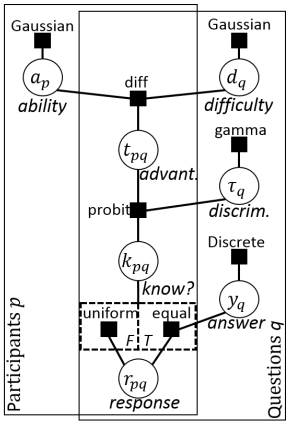


Figure 2. The DARE model in Tabular and factor-graph notation. The model is implemented in annotations to the three main tables Participants, Questions, and Responses. Tables QuestionsTrain and ResponsesTrain provide a mechanism for missing data.

Model	Language	LOC Data	LOC Model	LOC Inference	LOC total	Compile seconds	Infer seconds	Model log evidence	Avg. (log) prob. test responses.	Avg. (log) prob. test answers.
A	Tabular	0	17	0	17	126	10	-7499.74	(-1.432),0.239	(-3.435),0.032
A	Tabular II					0.41	1.47	-7499.74	(-1.432),0.239	(-3.424),0.033
A	Infer.NET	73	45	20	138	0.32	0.38	-7499.74	(-1.432),0.239	(-3.425),0.033
DA	Tabular	0	18	0	18	145	11	-5932.80	(-1.118),0.327	(-0.699),0.497
DA	Tabular II					0.40	1.54	-5933.52	(-1.118),0.327	(-0.739),0.478
DA	Infer.NET	73	47	21	141	0.34	0.43	-5933.25	(-1.118),0.327	(-0.724),0.485
DARE	Tabular	0	19	0	19	163	16	-5823.01	(-1.119),0.327	(-0.551),0.576
DARE	Tabular II					0.42	6.46	-5820.40	(-1.119),0.327	(-0.528),0.590
DARE	Infer.NET	73	49	22	144	0.37	2.8	-5820.40	(-1.119),0.327	(-0.528),0.590

Figure 3. Comparison of Tabular and direct Infer.NET implementations of different variants of the DARE model for multiple-choice questionnaires (machine configuration: DELL Precision T3600, Intel(R) Xeon(R) CPU E5-1620 with 16GB RAM, Windows 8 Enterprise and .NET 4.0). In all cases, the underlying algorithm is Infer.NET's Expectation Propagation. Tabular II gives the numbers for a direct translation from Tabular to Infer.NET that shows the performance issues of Fun (in particular the high compilation times) can be avoided.

ing the necessary data transformation code is much longer than the succinct and readable Tabular code that was added to the existing data schema to describe the same model. Tabular's excessively high compilation times are not due to the Tabular to Fun translation, which takes less than one second for each model, but to a flaw in the Fun compiler: Fun inlines all data before compiling, a convenient but unnecessary measure avoided by Infer.NET. To demonstrate that the excessive compile times can be reduced, we prototyped a second compiler, Tabular II, that translates Tabular programs directly to Infer.NET. On the DARE case study Tabular II improves compile times by two orders of magnitude, and inference time by up to one order of magnitude, yielding performance that is more competitive with handwritten Infer.NET (Figure 3).

7. Query-by-Missing-Value

Inference of latent columns requires that all output columns contain a valid value at each row. However, many real datasets contain missing values. *Query-by-missing-value* infers the posterior probability of missing values in output columns, conditioned on observed values actually present in the database. In a missing-values query, each attribute value is either *known*, or *missing*; we use ? to denote missing values.

Query-by-Missing-Value Database: $d^?$

$V^? ::= ? \mid V$	missing or known value
$r^? ::= \{c_1 = V_1^?, \ldots, c_n = V_n^?\}$	query-by-missing-value row
$R^? ::= [r_0^?; \ldots; r_n^?]$	query-by-missing-value table
$d^? ::= \{t_1 = R_1^?, \ldots, t_n = R_n^?\}$	query-by-missing-value database

Let a *missing-values learner* $(d_x, d_y^? \mid \mathbb{S})$ be a learner where d_x is a normal value and $d_y^?$ is a query-by-missing-value database. Such a learner can be queryable (as defined in Section 4.9), where we let $\Gamma \vdash ? : T$ for any T and Γ.

The result of inference on a queryable missing-values learner is the joint posterior distribution for all the ? entries in $d_y^?$, in addition to the latent columns and the parameters of each table. For a formal definition, we need to compute the observations of $d_y^?$, that is, the entries in $d_y^?$ present in the database and their values.

Observations of a missing-values query: $\mathsf{O}_E(\cdot)$

$$\mathsf{O}_E(?) \triangleq \textbf{true} \qquad \mathsf{O}_E(V) \triangleq E = V$$

$$\mathsf{O}_E(\{c_i = V_i^?\}^{i \in 1..n}) \triangleq \bigwedge_{i \in 1..n} \mathsf{O}_{E.c_i}(V_i^?)$$

$$\mathsf{O}_E([r_i^?]^{i \in 0..n}) \triangleq \bigwedge_{i \in 0..n} \mathsf{O}_{E[i]}(r_i^?)$$

$$\mathsf{O}_E(\{t_i = R_i^?\}^{i \in 1..n}) \triangleq \bigwedge_{i \in 1..n} \mathsf{O}_{E.c_i}(R_i^?)$$

Proposition 3. *If $L(d_x, d_y^? \mid \mathbb{S})$ is a queryable missing-values learner and $\mathbb{S} \Downarrow \langle E_h, (h)E_w, (h,w,x)E_{yz} \rangle$ then the prior distribution of L is given by P_E where $E = \textbf{let}\ h = \#(d_x)\ \textbf{in let}\ w = E_w\ \textbf{in let}\ x = d_x\ \textbf{in}\ w, E_{yz}$, and the joint posterior is the conditional probability distribution $\mathrm{P}_E\left[w, yz \mid \mathsf{O}_{\textbf{fst}\ yz}(d_y^?)\right]$.*

7.1 Example of Query-by-Missing-Value

Inferno is an experimental embedding of probabilistic inference in a spreadsheet (`http://research.microsoft.com/inferno/`). Given a probabilistic model for the whole spreadsheet, Inferno can fill in the missing values of empty cells, and also detect outliers: cells whose values are far from what is predicted by the model.

An Inferno spreadsheet can be considered as a queryable learner, where each spreadsheet column is an output but may have missing values, and there is an additional latent column for each row. The Tabular schema below corresponds to the Generalized Gaussian model produced by Inferno on a three-column table. We here consider only real-valued columns; other data types such as Booleans and integers can also be encoded as (vectors of) real numbers with appropriate (probabilistically invertible) link functions.

GG			
V	vector	latent	CVectorGaussian(Ncols=3)
X0	real	output	V[0]
X1	real	output	V[1]
X2	real	output	V[2]

(The library model CVectorGaussian is akin to CGaussian, but outputs vectors from a multivariate Gaussian distribution with Gaussian and Wishart priors.)

The query is a table *GG* containing the spreadsheet data, with empty cells replaced by ?, such as the following.

GG

ID	X0	X1	X2
0	1.0	2.1	2.9
1	2.1	?	6.3
2	?	2.7	3.5

Here $\mathsf{O}_y(GG) = y[0].\mathsf{X0} = 1.0 \wedge y[0].\mathsf{X1} = 2.1 \wedge \cdots \wedge y[2].\mathsf{X1} = 2.7 \wedge y[2].\mathsf{X2} = 3.5$.

7.2 Translating Query-by-Missing-Value to Query-by-Latent-Column

Missing-values queries can be answered by translating them to a latent column query and performing inference on the latter. The key idea is that for each output column of the original table we create a new table that contains just the known values in that column. In the translation of the orginal table, each output column is simply turned into a latent column. For example, the Inferno GG model translates to the following tables.

GG'			
V	vector	latent	CVectorGaussian(Ncols=3)
X0	real	latent	V[0]
X1	real	latent	V[1]
X2	real	latent	V[2]
X0			
R	link(GG')	input	
V	real	output	R.X0
X1			
R	link(GG')	input	
V	real	output	R.X1
X2			
R	link(GG')	input	
V	real	output	R.X2

Above, the query tables (X0, X1, and X2) each contain a value column V and a reference column R, which denotes the row from which the value came. Since the GG table contains no input columns, the translated GG' table contains only latent attributes, which do not show up in the query.

GG'

ID
0
1
2

All the data is in the query tables.

X0

ID	R	V
0	0	1.0
1	1	2.1

X1

	R	V
0	0	2.1
1	2	2.7

X2

	R	V
0	0	2.9
1	1	6.3
2	2	3.5

7.3 Formal Translation

We fix a queryable missing-values learner $L = (d_x, d_y^? \mid \mathbb{S})$ where $\mathbb{S} = (t_j \mapsto \mathbb{T}_j)^{j \in 1..m}$ and each table $\mathbb{T}_j = (c_{ji} \mapsto A_{ji} : T_{ji})^{i \in 1..n_j}$. Let the outputs $O_j \triangleq (i \in 1..n_j \mid A_{ji} = \textbf{output}(\_))$ be an ordered sequence of the output columns of table j for $j \in 1..m$. We assume fresh table names t_{ji} for $j \in 1..m$ and $i \in O_j$.

We turn output annotations into latent column annotations by letting $[\![\textbf{output}(M)]\!] \triangleq \textbf{latent}(M)$, and $[\![A]\!] \triangleq A$ otherwise. We extend $[\![\cdot]\!]$ to tables and schemas as follows.

$$[\![\mathbb{T}_j]\!] \triangleq (c_{ji} \mapsto [\![A_{ji}]\!] : T_{ji})^{i \in 1..n_j}$$

$$\mathbb{T}_{ji} \triangleq (\mathsf{R} \mapsto \textbf{input} : \textbf{int})$$
$$(\mathsf{V} \mapsto \textbf{output}(\mathsf{R} : \textbf{link}(t_j)).c_{ji} : T_{ji}) \text{ if } i \in O_j$$

$$[\![\mathbb{S}]\!] \triangleq (t_j \mapsto [\![\mathbb{T}_j]\!], (t_{ji} \mapsto \mathbb{T}_{ji})^{i \in O_j})^{j \in 1..m}$$

In the example above, GG' $= [\![\mathsf{GG}]\!]$, and the query tables Xi correspond to instances of $\mathbb{T}_{ji}$.

To translate the database, we first translate the observations in $d_y^?$.

$$Rx_{ji} \triangleq [\{\mathsf{R} = k\} \mid d_y^?.t_j[k].c_{ji} \neq ?]^{k \in 0..|d_y^?.t_j|-1}$$
$$Ry_{ji} \triangleq [\{\mathsf{V} = d_y^?.t_j[k].c_i\} \mid d_y^?.t_j[k].c_{ji} \neq ?]^{k \in 0..|d_y^?.t_j|-1}.$$

Here the contents of the query tables Xi above correspond to Rx_{ji}, Ry_{ji}.

The translations of the original tables (GG' above) have no observed values.

$$Ry_j \triangleq [\{\}]^{k \in 0..|d_x.t_j|-1}$$

Finally, we can combine these tables into a new database d'_x, d'_y. Here d'_x extends the inputs of d_x with the reference columns of the

query tables t_{ji}, while d'_y only has data in the query tables.

$$d'_x \triangleq \{t_j \mapsto d_x.t_j; \{t_{ji} \mapsto Rx_{ji}\}^{i \in O_j}\}^{j \in 1..m}$$
$$d'_y \triangleq \{t_j \mapsto Ry_j;\ \{t_{ji} \mapsto Ry_{ji}\}^{i \in O_j}\}^{j \in 1..m}$$

Lemma 4. *If $L = (d_x, d_y^? \mid \mathbb{S})$ is a queryable missing-values learner, then $(d'_x, d'_y \mid [\![\mathbb{S}]\!])$ as defined above is a queryable learner.*

To answer the missing-values query using the results of inference for the translated learner, we need to go from an inferred distribution for the translated schema $[\![\mathbb{S}]\!]$ to a distribution for the original schema $\mathbb{S}$. This is done by the function I defined below.

$$\begin{aligned} I(w,(\_,z)) = (\{t_j = w.t_j\}^{j \in 1..m}, \\ (\{t_j = [\{c_{ji} = z.t_j[k].c_{ji}\}^{i \in O_j}]^{k \in 0..|d_y^?.t_j|-1}\}^{j \in 1..m}, \\ \{t_j = [\{c_{ji} = z.t_j[k].c_{ji}\}^{i \in L_j}]^{k \in 0..|d_y^?.t_j|-1}\}^{j \in 1..m})). \end{aligned}$$

We can now show that the translation is correct: it reduces query-by-missing-value to query-by-latent-column.

Theorem 3 (Query-by-Missing-Value)**.**
Let $L = (d_x, d_y^? \mid \mathbb{S})$ be a queryable missing-values learner, and $L' = (d'_x, d'_y \mid [\![\mathbb{S}]\!])$ as defined above. If μ is (a version of) the semantics of the latent column query on $[\![L]\!]$ as given in Theorem 2 then $I^{-1}\mu$ is (a version of) the joint posterior conditional distribution $\mathrm{P}_E\left[w, yz \mid \mathsf{O}_{\mathbf{fst}\,yz}(d_y^?)\right]$ of L as given in Proposition 3.

Proof: See Appendix C. The proof idea is that compilation merely adds deterministic data and copies of random variables, which are then ignored by I. ■

As an optimization, an implementation may translate only those output columns where some data is actually missing to new tables. In the example above, there are no missing values in column X2 in the database, so it can remain observed in GG', and no new table needs to be created for its contents.

User/Movie/Rating Recommender Recall the User/Movie/Rating Schema of Section 3.3. Given existing tables of users, movies, and ratings, suppose we wish to recommend to user i movies that they are likely to rate with five stars. To do so, we first modify the annotation on the movie column of the Rating table, adding a uniform per-row prior distribution.

Rating			
u	**link**(User)	**input**	
m	**link**(Movie)	**output**	DiscreteUniform(**sizeof**(Movie))
Score	**int**	**output**	CDiscrete(N=5)[u.z,m.z]

We then add a single row $\{\mathsf{u} = i;\ \mathsf{m} = ?;\ \mathsf{Score} = 5\}$ to the existing data in the Rating table, denoting that user i has rated an unknown movie with 5 stars. This missing-values query is then translated to a corresponding latent column query in the manner defined above. Inference returns a discrete distribution over movie IDs for the missing value. Finally, high probability IDs can be selected for recommendation to the user.

In a variation of this query, we can weight the results by how many people have seen (that is, rated) each movie. To this end, we add interdependence between rows (a shared frequency prior) by instead using the model CDiscrete(N=**sizeof**(Movie)) for the movie column, and then proceed as above.

8. Related work

There has been previous work exploring the interface of databases and probabilistic inference. Specifically, we consider work on probabilistic programming languages, probabilistic databases, and statistical relational learning.

8.1 Probabilistic Programming Languages

There is by now a number of probabilistic programming languages, that differ in their target audience, expressive power, performance, and philosophy. BUGS (Bayesian Inference using Gibbs sampling) (Gilks et al. 1994) is a simple language for specifying probabilistic models that allows for inference using Gibbs sampling. It is widely used in the Bayesian community, but so far does not scale to large datasets. Microsoft Research's Infer.NET (Minka et al. 2012) achieves better scalabililty through support of deterministic approximate inference algorithms such as expectation propagation and variational message passing. Church (Goodman et al. 2008) is a relatively new probabilistic programming language based on Lisp, which allows for recursion and enables non-parametric Bayesian models through memoization. Furthermore, there are languages like IBAL (Pfeffer 2007) and Figaro (Pfeffer 2009), which incorporate decision-theoretic concepts as well. FACTORIE (McCallum et al. 2009) is an imperative framework for constructing graphical models in the form of factor graphs, used mostly for information extraction. All these languages follow the traditional paradigm of separating the code from the data schema and hence make it necessary to replicate the data schema within the language and to import the data from a database. On the other hand, Tabular is focused on learning from relational data, and does not directly address some of the emerging application areas of probabilistic programming such as vision as inverse graphics (Mansinghka et al. 2013; Wingate et al. 2011), or decision making for security (Mardziel et al. 2011).

8.2 Probabilistic Databases

Probabilistic databases represent a line of research in which the database community is concerned with the question of how to handle uncertain knowledge in relational databases (see, for example, Dalvi et al. (2009)). Typically, the assumption is made that each tuple is only in the database with a given probability, and that the presence of different tuples are independent events. The resulting probabilistic database can be interpreted in terms of the possible worlds semantics. It is further assumed that the probability values associated with each tuple are provided by the data collector, for example, from knowledge about measuring errors or from probabilistic models outside the probabilistic database. The main technical difficulty is to evaluate queries against probabilistic databases because despite the simplistic independence assumption on the presence of tuples, complex queries involving logical and aggregation operators can lead to difficult inference problems. This is also the main difference to the Tabular approach: whereas probabilistic databases work with concrete probabilities, Tabular works with non-probabilistic database schemas containing simple tuples (possibly with missing values) and allows building probabilistic models based on that data. In contrast to probabilistic database systems Tabular is thus compatible with the vast majority of existing relational datasets.

SimSQL (Cai et al. 2013) is a recent database system based on specifying, simulating, and querying database-valued Markov chains. SimSQL supports recursive definitions of stochastic tables, and can run in a MapReduce environment. It would be interesting to consider compilation of Tabular models and inference questions to SimSQL.

8.3 Statistical Relational Learning

Statistical Relational Learning operates in domains that exhibit both uncertainty and relational structure (see Getoor and Taskar (2007) for an excellent overview). Several contributions focus on combining probability and first-order logic, such as Bayesian Logic (BLOG) (Milch et al. 2005) which allows reasoning about unknown objects or Bayesstore (Wang et al. 2008), which bridges the world of probabilistic databases and statistical relational learning.

Tabular is more closely related to work that makes direct use of data in a relational database schema such as Getoor et al. (2007), Heckerman et al. (2007), and Neville and Jensen (2007). Tabular is based on directed graphical models, distinguishing it from Markov Logic (Domingos and Richardson 2004); there are several substantial implementations of Markov Logic including Alchemy (Kok et al. 2007), and Tuffy (Niu et al. 2011). Tabular was also inspired by a concept called PQL (Van Gael 2011) which augments the SQL query language with statements that construct a factor graph aligned with a given database schema. In summary, Tabular can be viewed as a language that enables the construction of statistical relational models directly from a schema, but goes beyond prior work in this field in that it allows the introduction of latent variables and models continuous as well as discrete variables.

Tabular was directly inspired by the question of finding a textual notation for the factor graphs generated by InfernoDB (Singh and Graepel 2012) which constructs a hierarchical mixture-based graphical model in Infer.NET (Minka et al. 2012) from an arbitrary relational schema. CrossCat (Shafto et al. 2006) is a related model, which handles single tables with mixed types (real, integer, bool). With Tabular, these types of model can be implemented in a few lines of code, and we envisage the automatic synthesis of a Tabular program that best models a given relational dataset, similar to the work of Grosse et al. (2012) on matrix decompositions.

9. Conclusions

We propose *schema-driven probabilistic programming* as a new principle of programming language design. The idea is to design a probabilistic modelling language by starting with a database schema and enriching it with notations for describing random variables, their probability distributions and interdependencies, how they relate to data matching the schema, and what is to be inferred.

Our design of Tabular is an instance of this principle, where the underlying schema is a typed relational model (subject to some restrictions), and where we confer semantics on Tabular schemas by using factor graphs as the underlying probabilistic model.

Acknowledgments

Conversations about this work with Chris Bishop, Lucas Bordeaux, John Bronskill, Tom Minka, and John Winn were invaluable. Misha Aizatulin made many contributions to the Fun system on which this work depends. William Cushing, Dylan Hutchison, Marcin Szymczak, Siddharth Srivastava, and Danny Tarlow commented on a draft. We would like to thank John Rust and Michal Kosinski from the Cambridge Psychometrics Centre as well as Pearson Assessments for providing the IQ dataset for research purposes.

References

N. L. Ackerman, C. E. Freer, and D. M. Roy. Noncomputable conditional distributions. In *LICS*, pages 107–116. IEEE Computer Society, 2011. ISBN 978-0-7695-4412-0.

Y. Bachrach, T. Graepel, T. Minka, and J. Guiver. How to grade a test without knowing the answers - a Bayesian graphical model for adaptive crowdsourcing and aptitude testing. In *Proceedings of the 29th International Conference on Machine Learning, ICML 2012*, 2012.

S. Bhat, J. Borgström, A. D. Gordon, and C. V. Russo. Deriving probability density functions from probabilistic functional programs. In *TACAS*, pages 508–522, 2013.

C. M. Bishop. Model-based machine learning. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 371(1984), 2013.

D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent Dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003.

J. Borgström, A. D. Gordon, M. Greenberg, J. Margetson, and J. V. Gael. Measure transformer semantics for Bayesian machine learning. *Logical Methods in Computer Science*, 9, abs/1308.0689(3), 2013. Preliminary version at ESOP'11.

Z. Cai, Z. Vagena, L. L. Perez, S. Arumugam, P. J. Haas, and C. M. Jermaine. Simulation of database-valued markov chains using simsql. In *SIGMOD Conference*, pages 637–648, 2013.

N. N. Dalvi, C. Ré, and D. Suciu. Probabilistic databases: diamonds in the dirt. *Commun. ACM*, 52(7):86–94, 2009.

P. Domingos and M. Richardson. Markov logic: A unifying framework for statistical relational learning. In *Proceedings of the ICML-2004 Workshop on Statistical Relational Learning and its Connections to other Fields*, pages 49–54, 2004.

L. Getoor and B. Taskar, editors. *Introduction to Statistical Relational Learning*. The MIT Press, 2007.

L. Getoor, N. Friedman, D. Koller, A. Pfeffer, and B. Taskar. Probabilistic relational models. In Getoor and Taskar (2007).

W. R. Gilks, A. Thomas, and D. J. Spiegelhalter. A language and program for complex Bayesian modelling. *The Statistician*, 43:169–178, 1994.

M. Giry. A categorical approach to probability theory. In B. Banaschewski, editor, *Categorical Aspects of Topology and Analysis*, volume 915 of *Lecture Notes in Mathematics*, pages 68–85. Springer, 1982.

N. Goodman, V. K. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum. Church: a language for generative models. In *Uncertainty in Artificial Intelligence (UAI'08)*, pages 220–229. AUAI Press, 2008.

A. D. Gordon, M. Aizatulin, J. Borgström, G. Claret, T. Graepel, A. Nori, S. Rajamani, and C. Russo. A model-learner pattern for Bayesian reasoning. In *POPL*, pages 403–416, 2013.

R. Grosse, R. Salakhutdinov, W. T. Freeman, and J. B. Tenenbaum. Exploiting compositionality to explore a large space of model structures. In *Proceedings of the 28th Conference on Uncertainty in Artificial Intelligence UAI2012*, pages 306–315, 2012.

P. Hanrahan. Analytic database technologies for a new kind of user: the data enthusiast. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 577–578. ACM, 2012.

D. Heckerman, C. Meek, and D. Koller. Probabilistic Entity-Relationship Models, PRMs, and Plate Models. In Getoor and Taskar (2007).

R. Herbrich, T. Minka, and T. Graepel. Trueskill^{tm}: A Bayesian skill rating system. In *Advances in Neural Information Processing Systems (NIPS'06)*, pages 569–576, 2006.

M. Izbicki. Algebraic classifiers: a generic approach to fast cross-validation, online training, and parallel training. In *Proceedings of the 29th International Conference on Machine Learning (ICML 2013)*, 2013.

O. Kiselyov and C. Shan. Embedded probabilistic programming. In *Conference on Domain-Specific Languages*, pages 360–384, 2009.

S. Kok, M. Sumner, M. Richardson, P. Singla, H. Poon, D. Lowd, and P. Domingos. The Alchemy system for statistical relational AI. Technical report, University of Washington, 2007. http://alchemy.cs.washington.edu.

D. Koller and N. Friedman. *Probabilistic Graphical Models*. The MIT Press, 2009.

U. D. Lago and M. Zorzi. Probabilistic operational semantics for the lambda calculus. *RAIRO - Theor. Inf. and Applic.*, 46(3):413–450, 2012.

D. Lunn, C. Jackson, N. Best, A. Thomas, and D. Spieghalter. *The BUGS Book*. CRC Press, 2013.

D. Maier. Representing database programs as objects. In F. Bancilhon and P. Buneman, editors, *DBPL*, pages 377–386. ACM Press / Addison-Wesley, 1987. ISBN 0-201-50257-7.

V. K. Mansinghka, T. D. Kulkarni, Y. N. Perov, and J. B. Tenenbaum. Approximate Bayesian image interpretation using generative probabilistic graphics programs. Available at `http://arxiv.org/abs/1307.0060`, 2013.

P. Mardziel, S. Magill, M. Hicks, and M. Srivatsa. Dynamic enforcement of knowledge-based security policies. In *Computer Security Foundations Symposium (CSF'11)*, pages 114–128, 2011.

A. McCallum, K. Schultz, and S. Singh. Factorie: Probabilistic programming via imperatively defined factor graphs. In *Advances in Neural Information Processing Systems (NIPS'09)*, pages 1249–1257, 2009.

B. Milch, B. Marthi, S. J. Russell, D. Sontag, D. L. Ong, and A. Kolobov. BLOG: Probabilistic models with unknown objects. In *Probabilistic, Logical and Relational Learning — A Further Synthesis*, 2005.

T. Minka. *A family of algorithms for approximate Bayesian inference*. PhD thesis, MIT, 2001.

T. Minka and J. M. Winn. Gates. In *Advances in Neural Information Processing Systems (NIPS'08)*, pages 1073–1080. MIT Press, 2008.

T. Minka, J. Winn, J. Guiver, and D. Knowles. Infer.NET 2.5, 2012. Microsoft Research Cambridge. http://research.microsoft.com/infernet.

J. Neville and D. Jensen. Relational dependency networks. *Journal of Machine Learning Research*, 8(8):653–692, 2007.

F. Niu, C. Ré, A. Doan, and J. W. Shavlik. Tuffy: Scaling up statistical inference in Markov Logic Networks using an RDBMS. *Very Large Databases (PVLDB)*, 4(6):373–384, 2011.

K. Nowicki and T. A. B. Snijders. Estimation and prediction for stochastic blockstructures. *J. Amer. Statist. Assoc.*, 96:1077–1087, 2001.

A. Pfeffer. The design and implementation of IBAL: A general-purpose probabilistic language. In Getoor and Taskar (2007).

A. Pfeffer. Figaro: An object-oriented probabilistic programming language. Technical report, Charles River Analytics, 2009.

N. Ramsey and A. Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *POPL*, pages 154–165, 2002.

P. Shafto, C. Kemp, V. Mansinghka, M. Gordon, and J. B. Tenenbaum. Learning cross-cutting systems of categories. In *Proceedings of the 28th Annual Conference of the Cognitive Science Society*, 2006.

S. Singh and T. Graepel. Compiling relational database schemata into probabilistic graphical models. *CoRR*, abs/1212.0967, 2012.

J. Van Gael. PQL—probabilistic query language. Blog post available at `http://jvangael.github.io/2011/05/12/pqla-probabilistic-query-language/`, May 2011.

D. Z. Wang, E. Michelakis, M. Garofalakis, and J. M. Hellerstein. Bayesstore: managing large, uncertain data repositories with probabilistic graphical models. *Proc. VLDB Endow.*, 1(1):340–351, Aug. 2008.

D. Wingate, N. D. Goodman, A. Stuhlmüller, and J. M. Siskind. Nonstandard interpretations of probabilistic programs for efficient inference. In *NIPS*, pages 1152–1160, 2011.

Structure of the Appendix

- Appendix A recapitulates the formal semantics of Fun.
- Appendix B gives the proof of Theorem 1 (Translation Preserves Typing).
- Appendix C gives the proof of Theorem 3 (Query-by-Missing-Value).
- Appendix D shows screenshots of our Tabular user interface.
- Appendix E lists additional models and code re the DARE case study.

A. Formal Semantics of Fun

As usual, for precision concerning probabilities over uncountable sets, we turn to measure theory. The interpretation of a type T is the measurable set $\mathbf{V}_T$ of closed values of type T (real numbers, integers etc.). We write $\mathscr{B}_T$ for the Borel-measurable sets of $\mathbf{V}_T$, defined using the standard (Euclidian) metric, and ranged over by A, B.

A measure μ over T is a function, from (measurable) subsets of $\mathbf{V}_t$ to the non-negative real numbers extended with ∞, that is countably additive, that is, $\mu(\varnothing) = 0.0$ and $\mu(\cup_i A_i) = \Sigma_i \mu(A_i)$ if $A_1, A_2, \ldots$ are pair-wise disjoint. The measure μ is called a probability measure if $\mu(\mathbf{V}_T) = 1.0$. If μ is a probability measure on T and $f : T \to U$, we let the (pushforward) probability measure $f^{-1}\mu(A) \triangleq \mu(f^{-1}(A))$. In this context f is often called a *random variable*.

We associate a default or *stock* measure λ_T to each type T, inductively defined as the counting measure on $\mathbb{Z}$ and $\{()\}$, the Lebesgue measure on $\mathbb{R}$, and the product of the measures for $\{f_1 : T_1; \ldots; f_n : T_n\}$ and $T[\,]$ (Array sizes in Fun are deterministic since arrays arise from array literals or for-comprehensions with a constant upper bound). If f is a non-negative (measurable) function $T \to \textsf{real}$, we let $\int f$ be the Lebesgue integral of f with respect to λ_T, if the integral is defined. This integral coincides with $\Sigma_{x \in \mathbf{V}_T} f(x)$ for discrete types T, and with the standard Riemann integral (if it is defined) on $T = \textsf{real}$. We also write $\int f(x)\, dx$ for $\int \lambda x.f(x)$, and $\int f(x)\, d\mu(x)$ for Lebesgue integration with respect to the measure μ on T. The Iverson brackets $[p]$ are 1.0 if predicate p is true, and 0.0 otherwise. We write $\int_A f$ for $\int \lambda x.[x \in A] \cdot f(x)$.

The semantics of a closed Fun expression E is a probability measure P_E over its return type. Open Fun expressions have a straightforward semantics (Ramsey and Pfeffer 2002) in the probability monad (Giry 1982). Below, σ is a substitution, that gives values to the free variables of E. When X is a term (possibly with binders), we write $x_1, \ldots, x_n \sharp X$ if none of the x_i appear free in X.

Monadic Semantics of Fun with arrays: $\mathscr{P}[\![E]\!]\ \sigma$

We assume that $z, z_1, \ldots, z_n \sharp E, F, F_1, E_1, \ldots, E_n, x, \sigma$.

$(\mu \gg\!= f)\ A \triangleq \int f(x)(A)\, d\mu(x)$ — Monadic bind

$(\texttt{return}\ V)\ A \triangleq 1$ if $V \in A$, else 0 — Monadic return

$$\mathscr{P}[\![x]\!]\ \sigma \triangleq \texttt{return}\ (x\sigma)$$
$$\mathscr{P}[\![s]\!]\ \sigma \triangleq \texttt{return}\ s$$
$$\mathscr{P}[\![[E_1; \ldots; E_n]]\!]\ \sigma \triangleq \mathscr{P}[\![E_1, \ldots, E_n]\!]^{\sigma}_{z}[\texttt{return}\ [z_1; \ldots; z_n]]$$
$$\mathscr{P}[\![[f_1 = E_1; \ldots; f_n = E_n]]\!]\ \sigma \triangleq \mathscr{P}[\![E_1, \ldots, E_n]\!]^{\sigma}_{z}[\texttt{return}\ \{f_1 = z_1; \ldots; f_n = z_n\}]$$
$$\mathscr{P}[\![E[F]]\!]\ \sigma \triangleq \mathscr{P}[\![E]\!]\ \sigma \gg\!= \lambda z.\mathscr{P}[\![F]\!]\ \sigma \gg\!= \lambda w.\texttt{return}\ z[w]$$
$$\mathscr{P}[\![\textsf{if}\ E\ \textsf{then}\ F_1\ \textsf{else}\ F_2]\!]\ \sigma \triangleq \mathscr{P}[\![E]\!]\ \sigma \gg\!= \lambda z.\texttt{if}\ z\ \texttt{then}\ \mathscr{P}[\![F_1]\!]\ \sigma\ \texttt{else}\ \mathscr{P}[\![F_2]\!]\ \sigma$$
$$\mathscr{P}[\![\textsf{let}\ x = E\ \textsf{in}\ F]\!]\ \sigma \triangleq \mathscr{P}[\![E]\!]\ \sigma \gg\!= \lambda z.\mathscr{P}[\![F]\!]\ (\sigma, x \mapsto z)$$
$$\mathscr{P}[\![[\textsf{for}\ x < E \to F]]\!]\ \sigma \triangleq \mathscr{P}[\![E]\!]\ (\sigma, x \mapsto 0) \gg\!= \lambda z_1.\mathscr{P}[\![E]\!]\ (\sigma, x \mapsto 1) \gg\!= \ldots \gg\!= \lambda z_c.\texttt{return}\ [z_1; \ldots; z_{E\sigma}]$$
$$\mathscr{P}[\![g(E_1, \ldots, E_n)]\!]\ \sigma \triangleq \mathscr{P}[\![E_1, \ldots, E_n]\!]^{\sigma}_{z}[\texttt{return}\ g(z_1, \ldots, z_n)]$$
$$\mathscr{P}[\![D(E_1, \ldots, E_n)]\!]\ \sigma \triangleq \mathscr{P}[\![E_1, \ldots, E_n]\!]^{\sigma}_{z}[\mu_{D(z_1, \ldots, z_n)}]$$
$$\mathscr{P}[\![E_1, \ldots, E_n]\!]^{\sigma}_{z}[\cdot] \triangleq \mathscr{P}[\![E_1]\!]\ \sigma \gg\!= \lambda z_1.\mathscr{P}[\![E_2]\!]\ \sigma \gg\!= \ldots \gg\!= \lambda z_{n-1}.\mathscr{P}[\![E_n]\!]\ \sigma \gg\!= \lambda z_n.[\cdot]$$

We let the semantics of a closed expression E be $\mathrm{P}_E \triangleq \mathscr{P}[\![E]\!]\ \varepsilon$, where ε denotes the empty substitution. We write $E \sim F$ if $\mathscr{P}[\![E]\!]\ \sigma = \mathscr{P}[\![F]\!]\ \sigma$ for all well-typed closing substitutions σ.

Definition 5 (Probability kernel)**.** *A function* $\kappa : \mathbf{V}_T \times \mathscr{B}_U \to [0, 1]$ *is called a* probability kernel *when*

(1) *for all* $B \in \mathscr{B}_U$*, the function* $\kappa(\cdot, B)$ *is measurable;*

(2) *for all $V \in \mathbf{V}_T$, the function $\kappa(V,\cdot)$ is a probability measure.*

Definition 6 (Conditional distribution)**.** *If μ is a probability measure on T and $f : T \to U_1$ and $g : T \to U_2$, then a probability kernel $\kappa : \mathbf{V}_{U_1} \times \mathscr{B}_{U_2} \to [0,1]$ is called a* version of the conditional probability distribution $\mu[g \mid f]$ *if for all $A \in \mathscr{B}_{U_1}$ and $B \in \mathscr{B}_{U_2}$,*

$$\mu\{\omega \mid f(\omega) \in A \wedge g(\omega) \in B\} = \int_A \kappa(x,B)\, d f^{-1}\mu(x).$$

Two versions of a conditional distribution may differ on a set of $f^{-1}\mu$-measure 0. However, two continuous versions must agree on the support of $f^{-1}\mu$, where the value $V \in \mathbf{V}_{U_1}$ is in the support of $f^{-1}\mu$ iff for all open sets $O \subseteq \mathbf{V}_{U_1}$ containing V we have $f^{-1}\mu(0) > 0$.

Lemma 7 ((Ackerman et al. 2011), Lemma 16)**.** *If κ_1, κ_2 are versions of the conditional probability distribution $\mu[g \mid f]$ as above, and both maps $x \mapsto \kappa_1(x,\cdot)$ and $x \mapsto \kappa_2(x,\cdot)$ are continuous at V, and V is in the support of $f^{-1}\mu$, then $\kappa_1(V,\cdot) = \kappa_2(V,\cdot)$.*

If $\vdash E : T_1 * \cdots * T_n$, and for $i = 1..m$ we have $x_1 : T_1, \ldots, x_n : T_n \vdash F_i : U_i$ where F_i contains no occurrence of $D(\cdot)$ and $\vdash V_i : U_i$, we write $\mathrm{P}_E[x_1,\ldots,x_n \mid F_1 = V_1 \wedge \cdots \wedge F_m = V_m]$ for a version κ of the regular conditional probability distribution $\mathrm{P}_E[\mathsf{id} \mid f]$ with f defined as the random variable $(x_1,\ldots,x_n) \mapsto (F_1,\ldots,F_m)$. This distribution is unique if $V = (V_1,\ldots,V_m)$ is a point of continuity of κ and also in the support of $f^{-1}\mathrm{P}_E$.

B. Proof of Theorem 1 (Translation Preserves Typing)

Our semantics translates Tabular schema, typed in Tabular contexts (declaring additional binding times and table typings), to Fun models whose components are typed in ordinary Fun contexts (simply relating variables to their types). The intuition behind our proof is to use binding times to extract the corresponding Fun contexts required to type check each of the three compartments (hyper, prior and gen) of the target model.

To this end, we define the following translation relation on contexts. The translation $\Gamma_{\text{Tabular}} \Downarrow_\ell \Gamma_{\text{Fun}}$ takes a Tabular context Γ_{Tabular} and binding time ℓ to produce an appropriately filtered Fun context Γ_{Fun} of variables available at, or before, the binding time ℓ. In addition, the translation expands table declarations, revealing their underlying array representation in Fun. A table identifier t is only accessible at binding time **xyz**, because t denotes the predictive database. (On the other hand, the identifier $\#t$ for the size of a table t is accessible at all binding times, because $\#t$ is introduced as a Tabular variable at binding time **h** by the rule (SCHEMA TABLE), and so if $\#t :^{\mathbf{h}} \mathbf{int}$ occurs in Γ_{Tabular} it is translated by (TRANS BIND LOWER) to $\#t : \mathbf{int}$ in Γ_{Fun}.)

Level-sensitive Translation of Contexts: $\Gamma_{\text{Tabular}} \Downarrow_\ell \Gamma_{\text{Fun}}$

(TRANS EMPTY)

$$\frac{}{\varnothing \Downarrow_\ell \varnothing}$$

(TRANS BIND LOWER)

$$\frac{\Gamma \Downarrow_\ell \Gamma' \quad \ell' \leq \ell}{\Gamma, x :^{\ell'} T \Downarrow_\ell \Gamma', x : T}$$

(TRANS BIND HIGHER)

$$\frac{\Gamma \Downarrow_\ell \Gamma' \quad \ell' > \ell}{\Gamma, x :^{\ell'} T \Downarrow_\ell \Gamma'}$$

(TRANS TABLE LOW)

$$\frac{\Gamma \Downarrow_\ell \Gamma' \quad \ell \leq \mathbf{w}}{\Gamma, t : \langle\{RT\}\rangle \Downarrow_\ell \Gamma'}$$

(TRANS TABLE HIGH)

$$\frac{\Gamma \Downarrow_{\mathbf{xyz}} \Gamma'}{\Gamma, t : \langle\{RT\}\rangle) \Downarrow_{\mathbf{xyz}} \Gamma', t : \{RT\}[]}$$

Lemma 8 (Totality)**.** *If $\Gamma \vdash \diamond$, then for all ℓ there is some Γ' such that $\Gamma \Downarrow_\ell \Gamma'$ and $\Gamma' \vdash \diamond$.*

Proof: By induction on the derivation of $\Gamma \vdash \diamond$. ■

Lemma 9 (Determinacy)**.** *If $\Gamma \Downarrow_\ell \Gamma_1$ and $\Gamma \Downarrow_\ell \Gamma_2$ then $\Gamma_1 = \Gamma_2$.*

Proof: By induction on the structure of Γ. ■

Lemma 10 (Domains)**.** *If $\Gamma \Downarrow_\ell \Gamma'$ then $\mathrm{dom}(\Gamma') \subseteq \mathrm{dom}(\Gamma)$.*

Proof: By induction on the derivation of $\Gamma \Downarrow_\ell \Gamma'$. ■

Lemma 11 (Variable Levelling)**.** *If $\Gamma_1, x :^\ell T, \Gamma_2 \vdash \diamond$ and $\ell \leq \ell'$ then there exist Γ'_1, Γ'_2 such that $\Gamma_1, x :^\ell T, \Gamma_2 \Downarrow_{\ell'} \Gamma'_1, x : T, \Gamma'_2$ and $\Gamma'_1, x : T, \Gamma'_2 \vdash \diamond$.*

Proof: By induction on the structure of Γ_2. ■

Lemma 12 (Table Levelling)**.** *If $\Gamma_1, t : \langle\{RT\}\rangle, \Gamma_2 \vdash \diamond$ then there exist Γ'_1, Γ'_2 such that $\Gamma_1, t : \langle\{RT\}\rangle, \Gamma_2 \Downarrow_{\mathbf{xyz}} \Gamma'_1, t : \{RT\}[], \Gamma'_2$ and $\Gamma'_1, t : \{RT\}[], \Gamma'_2 \vdash \diamond$.*

Proof: By induction on the structure of Γ_2. ■

Lemma 13 (Kind Preservation)**.** *If $\Gamma \vdash T$ and $\Gamma \Downarrow_\ell \Gamma'$ then $\Gamma' \vdash T$.*

Proof: By induction on the structure of Γ. ■

Lemma 14 (Monotonicity)**.** *If $\Gamma \vdash^\ell E : T$ and $\ell \leq \ell'$ then $\Gamma \vdash^{\ell'} E : T$.*

Proof: By induction on the derivation of $\Gamma \vdash^\ell E : T$. ■

The translation on schemas and tables produces intermediate Fun models that contain free variables with restricted binding times. The intermediate models are composed to produce a final, closed model. To prove correctness of the translation, we introduce an auxiliary typing judgment (really just a non-inductive predicate) that types an open Fun model in the Tabular context of its source program. The judgment has just one rule, and uses context translation to type check each compartment of the model in the derived Fun context available at that compartment's binding time.

Tabular Typing rules for Models: $\Gamma_{\text{Tabular}} \vdash P : Q$

(TABULAR MODEL)

$$\frac{\begin{array}{lll} \Gamma \Downarrow_{\mathbf{h}} \Gamma_0 & \Gamma_0 \vdash E_h : H & \mathrm{Det}(E_h) \\ \Gamma \Downarrow_{\mathbf{w}} \Gamma_1 & \Gamma_1, h : H \vdash E_w : W & \\ \Gamma \Downarrow_{\mathbf{xyz}} \Gamma_2 & \Gamma_2, h : H, w : W, x : X \vdash E : Y & \end{array}}{\Gamma \vdash \langle E_h, (h)E_w, (h,w,x)E\rangle : \langle H, W, X, Y\rangle}$$

We have that the relation $\vdash P : Q$ on closed models defined in Section 4.2 coincides with $\Gamma_{\text{Tabular}} \vdash P : Q$ when $\Gamma_{\text{Tabular}} = \varnothing$.

Proposition 15 (Coincidence for closed models)**.**
$\varnothing \vdash P : Q$ if and only if $\vdash P : Q$.

Proof: By (TRANS EMPTY) the translation of an empty Tabular context at any binding time is just an empty Fun context. Thus, when $\Gamma = \varnothing$, rule (TABULAR MODEL) collapses to (TYPE MODEL). ■

Lemma 16 (Open Translation Preserves Typing)**.**

(1) *If* $\Gamma \vdash^{\ell} E : T$ *then, for some* Γ', E'
- $\Gamma \Downarrow^{\ell} \Gamma'$
- $E \Downarrow E'$ *and*
- $\Gamma' \vdash E' : T$.

(2) *If* $\Gamma \vdash^{\ell} M : W, T$ *then, for some* $\langle E_w, (w)E\rangle$
- $M \Downarrow \langle E_w, (w)E\rangle$*; and*
- *for some* Γ_1, $\Gamma \Downarrow_{\boldsymbol{w}} \Gamma_1$ *and* $\Gamma_1 \vdash E_w : W$*; and*
- *for some* Γ_2, $\Gamma \Downarrow_l \Gamma_2$ *and* $\Gamma_2, w : W \vdash E : T$*;*

(3) *If* $\Gamma \vdash \mathbb{T} : Q$ *then, for some primitive model* P*,*
- $\mathbb{T} \Downarrow P$*; and*
- $\Gamma \vdash P : Q$.

(4) *If* $\Gamma \vdash \mathbb{S} : Q$ *then, for some primitive model* P*,*
- $\mathbb{S} \Downarrow P$*; and*
- $\Gamma \vdash P : Q$.

Proof: By induction on the typing judgments. ■

Restatement of Theorem 1 (Translation Preserves Typing)
If $\varnothing \vdash \mathbb{S} : Q$ *then there exists* P *such that* $\mathbb{S} \Downarrow P$ *and* $\vdash P : Q$.

Proof: Assume that $\varnothing \vdash \mathbb{S} : Q$. By Lemma 16 (Open Translation Preserves Typing) there exists P such that $\mathbb{S} \Downarrow P$ and $\varnothing \vdash P : Q$. By Proposition 15 (Coincidence for closed models) $\vdash P : Q$. ■

C. Proof of Theorem 3 (Query-by-Missing-Value)

We first expand the definitions of inference (Theorem 2 and Proposition 3) in the statement of the theorem.

Restatement of Theorem 3 (Query-by-Missing-Value):
Assume that $L = (d_x, d_y^? \mid \mathbb{S})$ *is a queryable missing values-learner such that* $\mathbb{S} \Downarrow \langle E_h, (h)E_w, (h,w,x)E_{yz}\rangle$*, and let*
$E \triangleq \textbf{let } h = \#(d_x) \textbf{ in let } w = E_w \textbf{ in let } x = d_x \textbf{ in } w, E_{yz}$.
Assume that L *translates to* $L' = (d'_x, d'_y \mid [\![\mathbb{S}]\!])$ *such that* $[\![\mathbb{S}]\!] \Downarrow \langle E'_h, (h)E'_w, (h,w,x)E'_{yz}\rangle$*, and let*
$E' \triangleq \textbf{let } h = \#(d'_x) \textbf{ in let } w = E'_w \textbf{ in let } x = d'_x \textbf{ in } w, E'_{yz}$.
Then $I^{-1}\mathrm{P}_{E'}\left[w, yz \mid \textbf{fst } yz = d'_y\right]$ *is a version of the joint posterior conditional distribution* $\mathrm{P}_E\left[w, yz \mid \mathsf{O}_{\textbf{fst } yz}(d_y^?)\right]$.

Below, we use the various symbols defined in the formal translation (Section 7.3). We first give helper lemmas relating the compilations of $\mathbb{S}$ and $[\![\mathbb{S}]\!]$. In these lemmas, we perform simple rewrites on Fun expressions, such as inlining of deterministic let-bindings, reordering of record fields, and partial evaluation of record field projection and array indexing (e.g., we rewrite $[\textbf{for } k < V \to E][i]$ to $E\,\{^i/_k\}$ when $i < V$).

The compilation of a translated table is the same as that of the original table, except that all observed attributes R_y are turned into latent ones and so appear in the second component of the return value of the sampling distribution instead of the first.

Lemma 17. *For* $j \in 1..m$*, if* $\mathbb{T}_j \Downarrow \langle E_h, (h)E_w, (h,w,x)\mathscr{L}(\{R_y\}, \{R_z\})\rangle)$ *then* $[\![\mathbb{T}_j]\!] \Downarrow \langle E_h, (h)E_w,\ (h,w,x)\mathscr{L}(\{\}, \{R_y; R_z\})\rangle)$.

Proof: By induction on $\mathbb{T}_j$. In the induction case $(c \mapsto A : T)\mathbb{T}$, the interesting case is when the annotation A is $\textbf{output}(M)$. Here we use rule (Trans Output) for A and (Trans Latent) for $[\![A]\!]$: they only differ in if c is added to R_y or to R_z. ■

The compilation of an observation table is as follows.

Lemma 18. *For all* $j \in 1..m$ *and* $i \in O_j$ *we have:*

$$\begin{aligned}
&\mathbb{T}_{ji} \Downarrow \langle \{\}, \\
&\quad (h)\textbf{let } \mathsf{V} = \{\} \textbf{ in } \{\mathsf{V}\$ = \mathsf{V}\}, \\
&\quad (h,w,x)\textbf{let } \mathsf{R} = x.\mathsf{R} \textbf{ in} \\
&\qquad \textbf{let } \mathsf{V} = (\textbf{let } w_v = w.\mathsf{V}\$ \textbf{ in } t_j[\mathsf{R}].c_{ji}) \textbf{ in} \\
&\qquad \{\mathsf{V} = \mathsf{V}\}, \{\}\rangle
\end{aligned}$$

Proof: By (Trans Input), (Trans Output), (Trans Simple) and (Trans Deref). ■

For $j \in 1..m$, we let $L_j \triangleq \{i \in 1..n_j \mid A_{ji} = \textbf{latent}(M)\}$ be the latent columns of table j. We write $\langle E_h, (h)E_w, (h,w,x)E_{yz}\rangle \sim \langle E'_h, (h)E'_w, (h,w,x)E'_{yz}\rangle$ iff $E_h \sim E'_h$ and $E_w \sim E'_w$ and $E_{yz} \sim E'_{yz}$.

In the compilation of the translated schema, inlining the let bindings corresponding to observation tables yields a simple correspondence to the compilation of the original schema. The universal quantification over R_w, R_y, R_z is in order to have a strong enough induction hypothesis: we only use this lemma with $R_w = R_y = R_z = \varnothing$.

Lemma 19. *If*

$$\begin{aligned}
\mathbb{S} \Downarrow \langle &\{\#t_j = 1\}^{j\in 1..m}, (h)\mathscr{L}_w[\{t_j = t_j\}^{j\in 1..m}], \\
&(h,w,x)\mathscr{L}_{yz}[\{t_j = E_{yj}\}^{j\in 1..m}, \{t_j = E_{zj}\}^{j\in 1..m}]\rangle
\end{aligned}$$

then there are $R'_w, R'_y, R'_z, \mathscr{L}'_w, \mathscr{L}'_{yz}$ *such that*

$$\begin{aligned}
[\![\mathbb{S}]\!] \Downarrow \langle &\{\#t_j = 1; R'_{hj}\}^{j\in 1..m}, (h)\mathscr{L}'_w[\{R'_w\}], \\
&(h,w,x)\mathscr{L}'_{yz}[\{R'_y\}, \{R'_z\}]\rangle
\end{aligned}$$

and for all R_w, R_y, R_z
with $\{t_{ji}, \#t_{ji} \mid j \in 1..m, i \in O_j\} \sharp R_w, R_y, R_z, \mathscr{L}_w, \mathscr{L}_{yz}$ *we have*

$$\begin{aligned}
\mathscr{L}'_w[\{R_w; R'_w\}] &\sim \mathscr{L}_w[\{R_w; (t_j = t_j; R'_{wj})^{j\in 1..m}\}] \\
\mathscr{L}'_{yz}[\{R_y; R'_y\}, \{R_z; R'_z\}] &\sim \mathscr{L}_{yz}[\{R_y; (t_j = E'_{yj}; R'_{yj})^{j\in 1..m}\}, \\
&\qquad \{R_z; (t_j = E'_{zj}; R'_{zj})^{j\in 1..m}\}]
\end{aligned}$$

where

$$\begin{aligned}
\{R'_{hj}\} &= \{\#t_{ji} = 1\}^{i\in O_j} \\
\{R'_{wj}\} &= \{t_{ji} = \{\mathsf{V}\$ = \{\}\}\}^{i\in O_j} \\
E'_{yj} &= [\textbf{for } k < h.\#t_j \to \{\}] \\
\{R'_{yj}\} &= \{t_{ji} = [\textbf{for } k < h.\#t_{ji} \to \{\mathsf{V} = t_j[x.t_{ji}[k].\mathsf{R}].c_{ji}\}]\}^{i\in O_j} \\
E'_{zj} &= [\textbf{for } k < h.\#t_j \to \{c_i = t_j[k].c_i\}^{i\in O_j \cup L_j}] \\
\{R'_{zj}\} &= \{t_{ji} = [\textbf{for } k < h.\#t_{ji} \to \{\}]\}^{i\in O_j}.
\end{aligned}$$

Proof: By induction on the schema. For the induction case $(t \mapsto \mathbb{T}_m)\mathbb{S}$, we have that
$[\![(t \mapsto \mathbb{T}_m)\mathbb{S}]\!] = (t_m \mapsto [\![\mathbb{T}_m]\!])(t_{mi} \mapsto \mathbb{T}_{mi})^{i\in O_m}[\![\mathbb{S}]\!]$.

We first show by induction on $|O_m|$ that

$$\begin{aligned}
(t_{mi} \mapsto \mathbb{T}_{mi})^{i\in O_m}[\![\mathbb{S}]\!] \Downarrow \langle &\{\#t_j = 1; R'_{hj}\}^{j\in 1..m}, (h)\mathscr{L}^m_w[\{R^m_w\}], \\
&(h,w,x)\mathscr{L}^m_{yz}[\{R^m_y\}, \{R^m_z\}]\rangle
\end{aligned}$$

such that for all R_w, R_y, R_z
with $\{t_{ji}, \#t_{ji} \mid j \in 1..m, i \in O_j\} \sharp R_w, R_y, R_z, \mathscr{L}_w, \mathscr{L}_{yz}$ we have

$$\begin{aligned}
\mathscr{L}^m_w[\{R_w; R^m_w\}] &\sim \mathscr{L}_w[\{R_w; R'_{wm}(t_j = t_j; R'_{wj})^{j\in 1..m-1}\}] \\
\mathscr{L}^m_{yz}[\{R_y; R^m_y\}, \{R_z; R^m_z\}] &\sim \mathscr{L}_{yz}[\{R_y; R'_{ym}(t_j = E'_{yj}; R'_{yj})^{j\in 1..m-1}\}, \\
&\qquad \{R_z; R'_{zm}(t_j = E'_{zj}; R'_{zj})^{j\in 1..m-1}\}]
\end{aligned}$$

The base case $O_m = \varnothing$ follows from the induction hypothesis of the outer induction. In the induction case $O_m = n, O'_m$, and we apply

rule (TRANS TABLE) to $(t_{mn} \mapsto \mathbb{T}_{mn}), (t_{mi} \mapsto \mathbb{T}_{mi})^{i \in O'_m}[\![\mathbb{S}]\!]$. The rule adds $\$t_{mn} = 1$ to the hyperparameter as desired.

Lemma 18 gives P_{mn} such that $\mathbb{T}_{mn} \Downarrow P_{mn}$. In the parameter,
$\textbf{let } t_{mn} = \textbf{let } h_t = \{\} \textbf{ in let } \mathsf{V} = \{\} \textbf{ in } \{\mathsf{V}\$ = \mathsf{V}\} \textbf{ in}$
$\textbf{let } \#t_{mn} = h.\#t_{mn} \textbf{ in}$
$\mathscr{L}_w^m[\{R_w; t_{mn} = t_{mn}; R_{wn}\}]$
$\sim$
$\mathscr{L}_w[\{R_w; t_{mn} = \{\mathsf{V}\$ = \{\}\}; R_{wn}\}]$ by inlining of deterministic **let**s since $t_{mn}, \#t_{mn}$ are fresh for $\mathscr{L}_w$ and R_w and R_{wn}.

In the gen-part, by inlining of deterministic **let**s we get $E_{t_{mn}} =$
$\textbf{let } h_t = \{\} \textbf{ in let } w_t = w.t \textbf{ in}$
$[\textbf{for } k < \#t_{mn} \to \textbf{let } x_k = x.t_{mn}[k] \textbf{ in}$
$\textbf{let } \mathsf{R} = x_k.\mathsf{R} \textbf{ in let } \mathsf{V} = \textbf{let } w_v = w.\mathsf{V}\$ \textbf{ in } t_m[\mathsf{R}].c_{mn} \textbf{ in}$
$\{\mathsf{R} = \mathsf{R}; \mathsf{V} = \mathsf{V}\}]$
$\sim$
$[\textbf{for } k < \#t_{mn} \to \{\mathsf{R} = x.t_{mn}[k].\mathsf{R}; \mathsf{V} = t_m[x.t_{mn}[k].\mathsf{R}].c_{mn}\}] =: E_{t_{mn}}$.
Similarly, since $t_{mn}, \#t_{mn}$ are fresh for $\mathscr{L}_{yz}$ and R_y, S_y and R_z, S_z we have
$\textbf{let } \#t_{mn} = h.\#t_{mn} \textbf{ in let } t_{mn} = E_{t_{mn}} \textbf{ in}$
$\mathscr{L}_{yz}^m[(\{R_y; t_{mn} = E_y; S_y\}, \{R_z; t = E_z; S_z\})]$
$\sim$
$\mathscr{L}_{yz}[(\{R_y; t_{mn} = E_y\sigma; S_y\}, \{R_z; t = E_z\sigma; S_z\})]$
where $\sigma = \{{}^{E_{t_{mn}}}/_{t_{mn}}\}\{{}^{h.\#t_{mn}}/_{\#t_{mn}}\}$ and $E_y\sigma = [\textbf{for } k < h.\#t_{mn} \to \{\}]$. Here $E_z\sigma \sim [\textbf{for } k < h.\#t_{mn} \to \{\mathsf{V} = t_m[x.t_{mn}[k].\mathsf{R}].c_{mn}\}]$ by partial evaluation of array indexing and record field projection.

The statement of the lemma then follows from Lemma 17 by applying rule (TRANS TABLE) to $(t_m \mapsto [\![\mathbb{T}_m]\!])(t_{mi} \mapsto \mathbb{T}_{mi})^{i \in O_m}[\![\mathbb{S}]\!]$. ■

We then let $\mathrm{Obs}_{ji} \triangleq (k \in 0..(|d_x.t_j| - 1) \mid d_y^?.t_j[k].c_i \neq ?)$ be the sequence of indexes of rows of table t_j that have an observed value in column c_i; we also define $n_{ji} \triangleq |\mathrm{Obs}_{ji}|$ as the number of observed entries in column i of table j. We let $Rz_{ji} \triangleq [\{\}]^{k \in 1..n_{ji}}$; recall that $Ry_j = [\{\}]^{k \in 1..|d_x.t_j|}$. To go from a pair of the priors and a predictive database for L to priors and database for the translated learner $(d'_x, d'_y, [\![\mathbb{S}]\!])$, we define $f(w,(y,z)) \triangleq (V_w, (V_y, V_z))$ where

$$V_w = \{t_j = w.t_j; (t_{ji} = \{\mathsf{V}\$ = \{\}\})^{i \in O_j}\}^{j \in 1..m}$$
$$V_y = \{t_j = Ry_j; (t_{ji} = [\{\mathsf{V} = y.t_j[k].c_i\}]^{k \in \mathrm{Obs}_{ji}})^{i \in O_j}\}^{j \in 1..m}$$
$$V_z = \{t_j = [\{c_i = E_{ijk}\}^{i \in O_j \cup L_j}]^{k \in 0..|d_x.t_j|-1}; (t_{ji} = Rz_{ji})^{i \in O_j}\}^{j \in 1..m}$$
$$E_{ijk} = \textbf{if } i \in O_j \textbf{ then } y.t_j[k].c_i \textbf{ else } z.t_j[k].c_i$$

Since L is queryable, we may assume that $\varnothing \vdash \mathbb{S} : \langle H, W, X, Y * Z \rangle$. The function I is an inverse of f.

Lemma 20. *$I \circ f = \mathsf{id}$ on $W * (Y * Z)$.*

Proof: I merely deletes the void values and copies of random variables added by f. ■

The lifting of f to distributions is parallel to the translation of Section 7.3.

Lemma 21. *$f^{-1}\mathrm{P}_E = \mathrm{P}_{E'}$.*

Proof: f merely adds the void values and copies of random variables that differ between the compilations of $\mathbb{S}$ and $[\![\mathbb{S}]\!]$ according to Lemma 19 with $R_x = R_y = R_z = \varnothing$. ■

To translate from the random variable that we condition on for L (i.e., a tuple of all observed values in $d^?$) to the conditioning RV for L' (i.e., the database d'_y) and back, we let l, j_k, r_k, i_k and V_k be given by the equation $\mathsf{O}_y(d_y^?) = \bigwedge_{k \in 1..l} y.t_{j_k}[r_k].c_{i_k} = V_k$, and define $V_y \triangleq (V_1, \ldots, V_l)$ and $g'(\_, (y, \_)) \triangleq (y.t_{j_1}[r_1].c_{i_1}, \ldots, y.t_{j_l}[r_l].c_{i_l})$ and

$$g(y_1, \ldots, y_l) \triangleq \{t_j = Ry_j;$$
$$(t_{ji} = [\{\mathsf{V} = y_{k_1}\}; \ldots; \{\mathsf{V} = y_{kn_{ji}}\} \mid 1 \leq k_1 < \cdots < k_{n_{ji}} \leq l \wedge$$
$$j = j_{k_1} = \cdots = j_{kn_{ji}} \wedge i = i_{k_1} = \cdots = i_{kn_{ji}}])^{i \in O_j}\}^{j \in 1..m}.$$

This technical lemma shows how the translations of observations relate to each other and to f.

Lemma 22.

(1) *$d'_y = g(V_y)$; and*
(2) *for all T_i, $g' \circ g = \mathsf{id}$ on $T_1 * \ldots T_l$; and*
(3) *$\textbf{fst} \circ \textbf{snd} \circ f = g \circ g'$ on $W * (Y * Z)$.*

Proof:

(1,2) Since the u such that $d_y^?.c_j[u].c_i \neq ?$ are precisely those where there is $k \in 1..l$ with $u = r_k$ and $j_k = j$ and $i_k = i$.

(3) Here

$$(\textbf{fst} \circ \textbf{snd} \circ f)(\_, (y, \_)) =$$
$$\{t_j = Ry_j; (t_{ji} = [\{\mathsf{V} = y.t_j[k].c_i\}]^{k \in \mathrm{Obs}_{ji}})^{i \in O_j}\}^{j \in 1..m} =$$
$$(g \circ g')(\_, (y, \_)).$$

■

Finally, we can complete the proof of Theorem 3 (Query-by-Missing-Value).

Proof: (of Theorem 3 (Query-by-Missing-Value)) We then have

$\mathrm{P}_E\left[w, yz \mid \mathsf{O}_{\textbf{fst}\,yz}(d_y^?)\right] =$	(By desugaring)
$\mathrm{P}_E\left[\mathsf{id} \mid g' = V_y\right] =$	(By Lemma 22(2))
$\mathrm{P}_E\left[\mathsf{id} \mid g' = (g' \circ g)(V_y)\right] =$	(By Lemma 22(3))
$\mathrm{P}_E\left[\mathsf{id} \mid \textbf{fst} \circ \textbf{snd} \circ f = g(V_y)\right] =$	(By Lemma 22(1))
$\mathrm{P}_E\left[\mathsf{id} \mid \textbf{fst} \circ \textbf{snd} \circ f = d'_y\right] =$	(By Lemma 20)
$\mathrm{P}_E\left[I \circ f \mid \textbf{fst} \circ \textbf{snd} \circ f = d'_y\right] =$	(By definition of $\cdot^{-1}\mu$)
$I^{-1}(\mathrm{P}_E\left[f \mid \textbf{fst} \circ \textbf{snd} \circ f = d'_y\right]) =$	(By Lemma 21)
$I^{-1}(\mathrm{P}_{E'}\left[\mathsf{id} \mid \textbf{fst} \circ \textbf{snd} = d'_y\right]) =$	(By sugaring)
$I^{-1}(\mathrm{P}_{E'}\left[w, yz \mid \textbf{fst}\ yz = d'_y\right])$.	

■

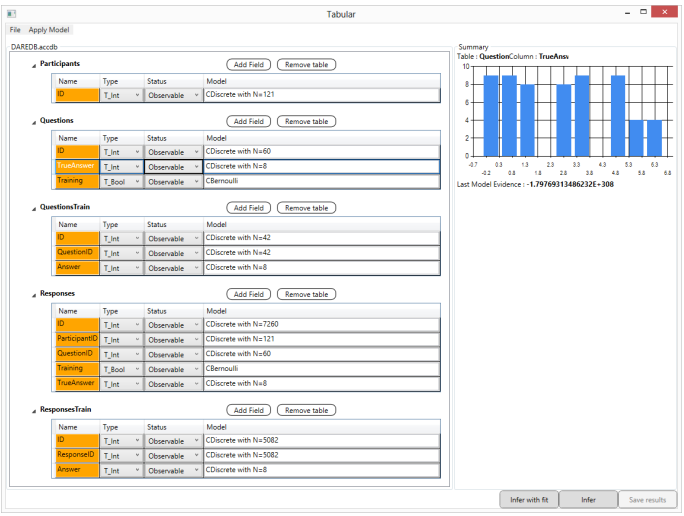


Figure 4. Step 1: user loads DARE database; application infers a default model from the schema and data.

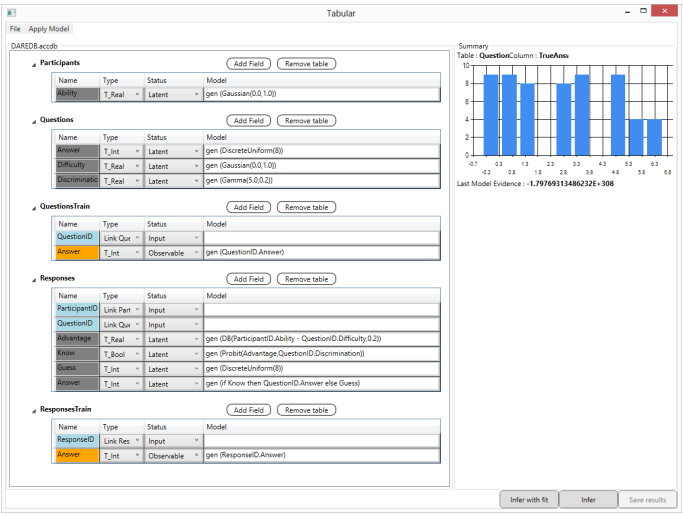


Figure 5. Step 2: user authors the DARE schema.

D. Application Screenshots

Figures 4-8 illustrate a user using our Tabular application to import a database, model it, and then examine the fruits of inference that are saved back to the database.

E. Case Study

Figures 9 and 10 depict the simpler A and DA variants of the DARE model (c.f. Figure 2) described in Section 6.

Figure 11 depicts the Infer.NET code to construct and run the DARE model.

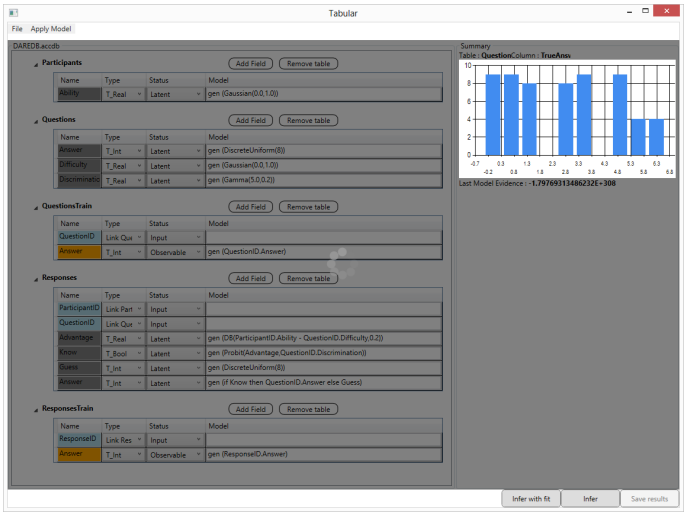


Figure 6. Step 3: user triggers inference.

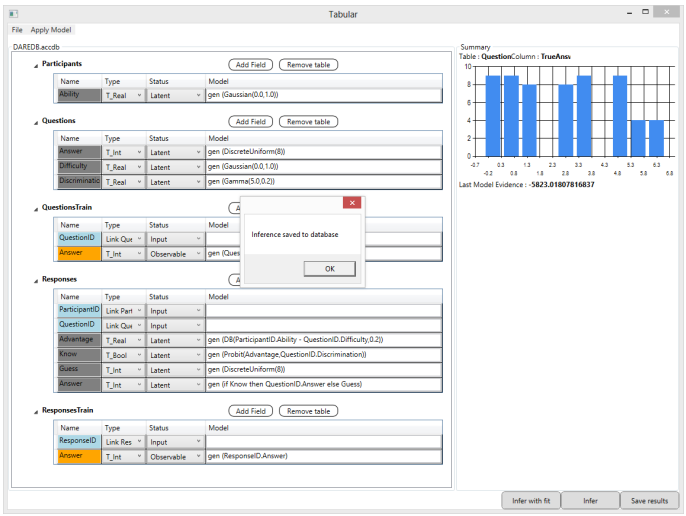


Figure 7. Step 4: user saves results back to database.

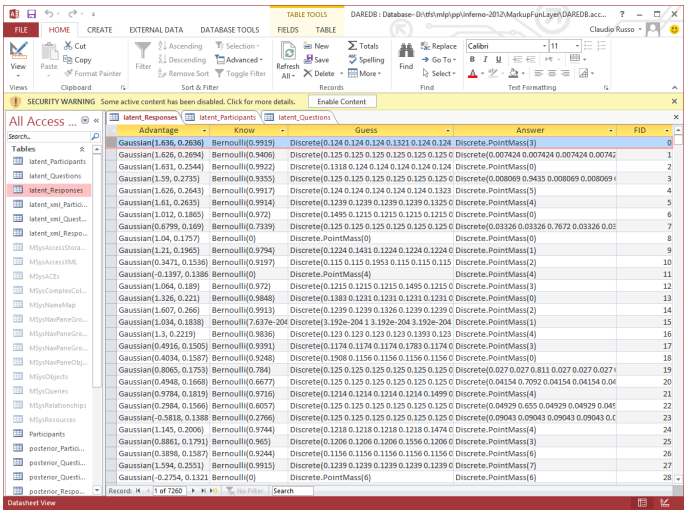


Figure 8. Step 5: user examines database with Microsoft Access; it now contains additional tables for latents and posteriors.

Participants			
Ability	**real**	**latent**	Gaussian(0,1)
Questions			
Answer	**int**	**latent**	Discrete(8)
QuestionsTrain			
QuestionID	**link**(Questions)	**input**	
Answer	**int**	**output**	QuestionID.Answer
Responses			
ParticipantID	**link**(Participants)	**input**	
QuestionID	**link**(Questions)	**input**	
Advantage	**real**	**latent**	DB(ParticipantID.Ability,0.2)
Know	**bool**	**latent**	Probit(Advantage,1)
Guess	**int**	**latent**	Discrete(8)
Response	**int**	**latent**	**if** Know **then** QuestionID.Answer **else** Guess
ResponsesTrain			
ResponseID	**link**(Responses)	**input**	
Response	**int**	**output**	ResponseID.Response

Figure 9. The Ability model (A)

Participants			
Ability	**real**	**latent**	Gaussian(0,1)
Questions			
Answer	**int**	**latent**	Discrete(8)
Difficulty	**real**	**latent**	Gaussian(0,1)
QuestionsTrain			
QuestionID	**link**(Questions)	**input**	
Answer	**int**	**output**	QuestionID.Answer
Responses			
ParticipantID	**link**(Participants)	**input**	
QuestionID	**link**(Questions)	**input**	
Advantage	**real**	**latent**	DB(ParticipantID.Ability − QuestionID.Difficulty,0.2)
Know	**bool**	**latent**	Probit(Advantage,1)
Guess	**int**	**latent**	Discrete(8)
Response	**int**	**latent**	**if** Know **then** QuestionID.Answer **else** Guess
ResponsesTrain			
ResponseID	**link**(Responses)	**input**	
Response	**int**	**output**	ResponseID.Response

Figure 10. The Difficulty-Ability model (DA)

```
public static void CreateAndRunDAREModel(
    Gaussian abilityPrior , Gaussian difficultyPrior , Gamma discriminationPrior,
    int nParticipants, int nQuestions, int nChoices,
    int[] participantOfResponse, int[] questionOfResponse,
    int[] response, int[] trainingResponseIndices,
    int[] answer, int[] trainingQuestionIndices)
{   // Model
    var Evidence = Variable.Bernoulli(0.5).Named("evidence");
    var EvidenceBlock = Variable.If(Evidence);
    var NQuestions = Variable.New<int>().Named("nQuestions");
    var NParticipants = Variable.New<int>().Named("nParticipants");
    var NChoices = Variable.New<int>().Named("nChoices");
    var NResponses = Variable.New<int>().Named("nResponses");
    var NTrainingResponses = Variable.New<int>().Named("nTrainingResponses");
    var NTrainingQuestions = Variable.New<int>().Named("nTrainingQuestions");
    var p = new Range(NParticipants).Named("p");
    var q = new Range(NQuestions).Named("q");
    var c = new Range(NChoices).Named("c");
    var n = new Range(NResponses).Named("n");
    var tr = new Range(NTrainingResponses).Named("tr");
    var tq = new Range(NTrainingQuestions).Named("tq");
    var AnswerOfQuestion = Variable.Array<int>(q).Named("answer");
    AnswerOfQuestion[q] = Variable.DiscreteUniform(c).ForEach(q);
    var QuestionOfResponse = Variable.Array<int>(n).Named("questionOfResponse");
    var ParticipantOfResponse = Variable.Array<int>(n).Named("participantOfResponse");
    var AnswerOfResponse = Variable.Array<int>(n).Named("response");
    QuestionOfResponse.SetValueRange(q);
    ParticipantOfResponse.SetValueRange(p);
    AnswerOfResponse.SetValueRange(c);
    var Know = Variable.Array<bool>(n).Named("know");
    var Ability = Variable.Array<double>(p).Named("ability");
    Ability [p] = Variable.Random(abilityPrior).ForEach(p);
    var Difficulty = Variable.Array<double>(q).Named("difficulty");
    Difficulty [q] = Variable.Random(difficultyPrior).ForEach(q);
    var Discrimination = Variable.Array<double>(q).Named("discrimination");
    Discrimination [q] = Variable.Random(discriminationPrior).ForEach(q);
    using (Variable.ForEach(n))
    {
        var advantage = (Ability[ParticipantOfResponse[n]] - Difficulty[QuestionOfResponse[n]]).Named("advantage");
        var advantageDamped = Variable<double>.Factor<double, double>(Damp.Backward, advantage, 0.2).Named("advantageDamped");
        var advantageNoisy = Variable.Gaussian(advantageDamped, Discrimination[QuestionOfResponse[n]]).Named("advantageNoisy");
            Know[n] = (advantageNoisy > 0);
        using (Variable.If(Know[n]))
            AnswerOfResponse[n] = AnswerOfQuestion[QuestionOfResponse[n]];
        using (Variable.IfNot(Know[n]))
            AnswerOfResponse[n] = Variable.DiscreteUniform(c);
    }
    var TrainingResponseIndices = Variable.Array<int>(tr).Named("trainingResponseIndices");
    TrainingResponseIndices.SetValueRange(n);
    var ObservedResponseAnswer = Variable.Array<int>(tr).Named("observedResponseAnswer");
    ObservedResponseAnswer[tr] = AnswerOfResponse[TrainingResponseIndices[tr]];
    var TrainingQuestionIndices = Variable.Array<int>(tq).Named("trainingQuestionIndices");
    TrainingQuestionIndices.SetValueRange(q);
    var ObservedQuestionAnswer = Variable.Array<int>(tq).Named("observedQuestionAnswer");
    ObservedQuestionAnswer[tq] = AnswerOfQuestion[TrainingQuestionIndices[tq]];
    EvidenceBlock.CloseBlock();
    // Hook up the data and run inference
    var nResponse = response.Length;
    NQuestions.ObservedValue = nQuestions;
    NParticipants.ObservedValue = nParticipants;
    NChoices.ObservedValue = nChoices;
    NResponses.ObservedValue = nResponse;
    var nTrainingResponses = trainingResponseIndices.Length;
    NTrainingResponses.ObservedValue = nTrainingResponses;
    var nTrainingQuestions = trainingQuestionIndices.Length;
    NTrainingQuestions.ObservedValue = nTrainingQuestions;
    ParticipantOfResponse.ObservedValue = participantOfResponse;
    QuestionOfResponse.ObservedValue = questionOfResponse;
    TrainingResponseIndices.ObservedValue = trainingResponseIndices;
    ObservedResponseAnswer.ObservedValue = Util.ArrayInit(nTrainingResponses, i => response[trainingResponseIndices[i]]);
    TrainingQuestionIndices.ObservedValue = trainingQuestionIndices;
    ObservedQuestionAnswer.ObservedValue = Util.ArrayInit(nTrainingQuestions, i => answer[trainingQuestionIndices[i]]);
    var Engine = new InferenceEngine() { ShowTimings = true, ShowWarnings = false, ShowProgress = false, NumberOfIterations = 10 };
    var AnswerOfResponsePosterior = Engine.Infer<Discrete[]>(AnswerOfResponse);
    var AnswerOfQuestionPosterior = Engine.Infer<Discrete[]>(AnswerOfQuestion);
    var LogEvidence = Engine.Infer<Bernoulli>(Evidence).LogOdds;
    var AbilityPosterior = Engine.Infer<Gaussian[]>(Ability);
    var DifficultyPosterior = Engine.Infer<Gaussian[]>(Difficulty);
    var DiscriminationPosterior = Engine.Infer<Gamma[]>(Discrimination);
}
```

Figure 11. Infer.NET DARE model, including model construction and inference code (compare to Figure 2).