Probabilistic Programs as Spreadsheet Queries[⋆]

Andrew D. Gordon[1,2], Claudio Russo[1], Marcin Szymczak[2], Johannes Borgström[3], Nicolas Rolland[1], Thore Graepel[1], and Daniel Tarlow[1]

[1]Microsoft Research, Cambridge, United Kingdom
[2]University of Edinburgh, Edinburgh, United Kingdom
[3]Uppsala University, Uppsala, Sweden

Abstract. We describe the design, semantics, and implementation of a probabilistic programming language where programs are spreadsheet queries. Given an input database consisting of tables held in a spreadsheet, a query constructs a probabilistic model conditioned by the spreadsheet data, and returns an output database determined by inference. This work extends probabilistic programming systems in three novel aspects: (1) embedding in spreadsheets, (2) dependently typed functions, and (3) typed distinction between random and query variables. It empowers users with knowledge of statistical modelling to do inference simply by editing textual annotations within their spreadsheets, with no other coding.

1 Spreadsheets and Typeful Probabilistic Programming

Probabilistic programming systems [11, 14] enable a developer to write a short piece of code that models a dataset, and then to rely on a compiler to produce efficient inference code to learn parameters of the model and to make predictions. Still, a great many of the world's datasets are held in spreadsheets, and accessed by users who are not developers. How can spreadsheet users reap the benefits of probabilistic programming systems?

Our first motivation here is to describe an answer, based on an overhaul of Tabular [13], a probabilistic language based on annotating the schema of a relational database. The original Tabular is a standalone application that runs fixed queries on a relational database (Microsoft Access). We began the present work by re-implementing Tabular within Microsoft Excel, with the data and program held in spreadsheets.

The conventional view is that the purpose of a probabilistic program is to define the random variables whose marginals are to be determined (as in the query-by-missing-value of original Tabular). In our experience with spreadsheets, we initially took this view, and relied on Excel formulas, separate from the probabilistic program, for post-processing tasks such as computing the mode (most likely value) of a distribution, or deciding on an action (whether or not to place a bet, say). We found, to our surprise, that combining Tabular models and Excel formulas is error-prone and cumbersome, particularly when the sizes of tables change, the parameters of the model change, or we simply need to update a formula for every row of a column.

In response, our new design contributes the principle that a probabilistic program defines a pseudo-deterministic query on data. The query is specified in terms of three sorts

[⋆] This work was supported by Microsoft Research through its PhD Scholarship Programme.

© Springer-Verlag Berlin Heidelberg 2015
J. Vitek (Ed.): ESOP 2015, LNCS 9032, pp. 1–25, 2015.
DOI: 10.1007/978-3-662-46669-8_1

of variable: (1) deterministic variables holding concrete input data; (2) nondeterministic random variables constituting the probabilistic model conditioned on input data; and (3) pseudo-deterministic query variables defining the result of the program (instead of using Excel formulas). Random variables are defined by draws from a set of builtin distributions. Query variables are defined via an **infer** primitive that returns the marginal posterior distributions of random variables. For instance, given a random variable of Boolean type, **infer** returns the probability p that the variable is **true**. In theory, **infer** is deterministic—it has an exact semantics in terms of measure theory; in practice, **infer** (and hence the whole query) is only pseudo-deterministic, as implementations almost always perform approximate or nondeterministic inference. We have many queries as evidence that post-processing can be incorporated into the language.

Our second motivation is to make a case for *typeful probabilistic programming* in general, with evidence from our experience of overhauling Tabular for spreadsheets. Cardelli [5] identifies the programming style based on widespread use of mechanically-checked types as *typeful programming*. Probabilistic languages that are embedded DSLs, such as HANSEI [17], Fun [2], and Factorie [19], are already typeful in that they inherit types from their host languages, while standalone languages, such as BUGS [9] or Stan [28], have value-indexed data schemas (but no user-defined functions). Still, we find that more sophisticated forms of type are useful in probabilistic modelling.

We make two general contributions to typeful probabilistic programming.
(1) *Value-indexed function types usefully organise user-defined components, such as conjugate pairs, in probabilistic programming languages.*

We allow value indexes in types to indicate the sizes of integer ranges and of array dimensions. We add value-indexed function types for user-defined functions, with a grid-based syntax. The paper has examples of user-defined functions (such as Action in Section 6) showing their utility beyond the fixed repertoire of conjugate pairs in the original Tabular. An important difficulty is to find a syntax for functions and their types that fits with the grid-based paradigm of spreadsheets.
(2) *A type-based information-flow analysis usefully distinguishes the stochastic and deterministic parts of a probabilistic program.*

To track the three sorts of variable, each type belongs to a *space* indicating whether it is: (**det**) deterministic input data, (**rnd**) a non-deterministic random variable defining the probabilistic model of the data, or (**qry**) a pseudo-deterministic query-variable defining a program result. Spaces allow a single language to define both model and query, while the type system governs flows between the spaces: data flows from **rnd** to **qry** via **infer**, but to ensure that a query needs only a single run of probabilistic inference, there are no flows from **qry** to **rnd**. There is an analogy between our spaces and levels in information flow systems: **det**-space is like a level of trusted data; **rnd**-space is like a level of untrusted data that is tainted by randomness; and **qry** is like a level of trusted data that includes untrusted data explicitly endorsed by **infer**.

The benefits of spaces include: (1) to document the role of variables, (2) to slice a program into the probabilistic model versus the result query, and (3) to prevent accidental errors. For instance, only variables in **det**-space may appear as indexes in types to guarantee that our models can be compiled to the finite factor graphs supported by inference backends such as Infer.NET [20].

This paper defines the syntax, semantics, and implementation of a new, more typeful Tabular. Our implementation is a downloadable add-in for Excel. For execution on data in a spreadsheet, a Tabular program is sliced into (1) an Infer.NET model for inference, and (2) a C# program to compute the results to be returned to the spreadsheet.

The original semantics of Tabular uses the higher-order model-learner pattern [12], based on a separate metalanguage. Given a Tabular schema $\mathbb{S}$ and an input database DB that matches $\mathbb{S}$, our semantics consists of two algorithms.

(1) An algorithm $\mathsf{CoreSchema}(\mathbb{S})$ applies a set of source-to-source reductions on $\mathbb{S}$ to yield $\mathbb{S}'$, which is in a core form of Tabular without user-defined functions and some other features.
(2) An algorithm $\mathsf{CoreQuery}(\mathbb{S}',DB)$ first constructs a probabilistic model based on the **rnd**-space variables in $\mathbb{S}'$ conditioned by DB, and then evaluates the **qry**-space variables in $\mathbb{S}'$ to assemble an output database DB'.

Our main technical results about the semantics are as follows.

(1) Theorem 1 establishes that $\mathsf{CoreSchema}(\mathbb{S})$ yields the unique core form $\mathbb{S}'$ of a well-typed schema $\mathbb{S}$, as a corollary of standard properties of our reduction relation with respect to the type system (Proposition 1, Proposition 2, and Proposition 3).
(2) Theorem 2 establishes pre- and post-conditions of the input and output databases when $DB' = \mathsf{CoreQuery}(\mathbb{S}',DB)$.

Beyond theory, the paper describes many examples of the new typeful features of Tabular, including a detailed account of Bayesian Decision Theory, an important application of probabilistic programming, not possible in the original form of Tabular. A language like IBAL or Figaro allows for rational decision-making, but via decision-specific language features, rather than in the core expression language. We present a numeric comparison of a decision theory problem expressed in Tabular versus the same problem expressed in C# with direct calls to Infer.NET, showing that we pay very little in performance in return for a much more succinct spreadsheet program.

2 Functions and Queries, by Example

Primer: Discrete and Dirichlet Distributions. To begin to describe the new features of Tabular, we recall a couple of standard distributions. If array $\mathsf{V} = [p_0;\ldots;p_{n-1}]$ is a *probability vector* (that is, each p_i is a probability and they sum to 1) then $\mathsf{Discrete}[n](\mathsf{V})$ is the discrete distribution that yields a sample $i \in 0..n-1$ with probability p_i. The distribution $\mathsf{Discrete}[2]([\frac{1}{2};\frac{1}{2}])$ models a coin that we know to be fair. If we are uncertain whether the coin is fair, we need a distribution on probability vectors to represent our uncertainty. The distribution $\mathsf{Dirichlet}[n]([c_0;\ldots;c_{n-1}])$ on a probability vector V represents our uncertainty after observing a count $c_i - 1$ of samples of i from $\mathsf{Discrete}[n](\mathsf{V})$ for $i \in 0..n-1$. We omit the formal definition, but discuss the case $n = 2$.

A probability vector V drawn from $\mathsf{Dirichlet}[2]([t+1;h+1])$ represents our uncertainty about the bias of a coin after observing t tails and h heads. It follows that $\mathsf{V} = [1-p;p]$ where p is the probability of heads. The expected value of p is $\frac{h+1}{t+h+2}$, and the variance of p diminishes as t and h increase. If $t = h = 0$, the expected value is $\frac{1}{2}$ and p is uniformly distributed on the unit interval. If $t = h = 10$ say, the expected value remains $\frac{1}{2}$ but p is much more likely near the middle than the ends of the interval.

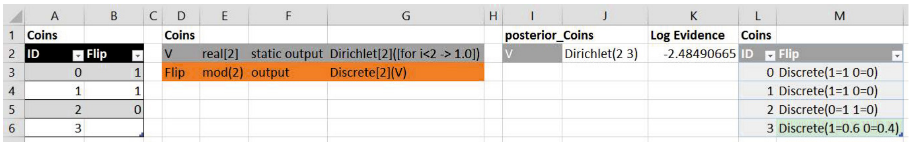
Review: Probabilistic Schemas in Tabular. Suppose we have a table named Coins with a column Flip containing a series of coin flips and wish to infer the bias of the coin. (The syntax [**for** i $<$ 2 $\rightarrow$1.0] is an array comprehension, in this case returning $[1.0, 1.0]$.)

table Coins (original Tabular)

V	**real**[]	**static output**	Dirichlet[2]([**for** i $<$ 2 $\rightarrow$1.0])
Flip	**int**	**output**	Discrete[2](V)

The model above (in original Tabular up to keyword renaming) is read as a probabilistic recipe for generating the coin flips from the unknown parameter V, conditioned on the actual dataset. The first line creates a random variable V $= [1-p; p]$ from Dirichlet$[2]([1;1])$, which amounts to choosing the probability p of heads uniformly from the unit interval. The second line creates a random variable Flip from Discrete$[n](V)$ for each row of the table and conditions the variable in each row to equal the actual observed coin flip, if it is present. Each Tabular variable is either at **static**- or **inst**-level. A **static**-variable occurs just once per table, whereas an **inst**-variable occurs for each row of the table. The default level is **inst**, so Flip is at **inst**-level.

Now, suppose the data for the column Flip is $[1;1;0]$; the prior distribution of V is updated by observing 2 heads and 1 tails, to yield the posterior Dirichlet$[2]([2;3])$, which has mean $\frac{3}{5}$. Given our example model, the fixed queries of this initial form of Tabular compute the posterior distribution of V, and write the resulting distributions as strings into the spreadsheet, as shown below. The missing value in cell B6 of the Flip column is predicted by the distribution in cell M6: 60% chance of 1, 40% chance of 0. (Cells E2 and E3 show dependent types of our new design, not of the original Tabular.)



	A	B	C	D	E	F	G	H	I	J	K	L	M
1	Coins			Coins					posterior_Coins		Log Evidence	Coins	
2	ID	Flip		V	real[2]	static output	Dirichlet[2]([for i<2 -> 1.0])		V	Dirichlet(2 3)	-2.48490665	ID	Flip
3	0	1		Flip	mod(2)	output	Discrete[2](V)					0	Discrete(1=1 0=0)
4	1	1										1	Discrete(1=1 0=0)
5	2	0										2	Discrete(0=1 1=0)
6	3											3	Discrete(1=0.6 0=0.4)

New Features of Tabular. Our initial experience with the re-implementation shows that writing probabilistic programs in spreadsheets is viable but suggests three new language requirements, explained in the remainder of this section.

(1) User-defined functions for abstraction (to generalize the fixed repertoire of primitive models in the original design).
(2) User-defined queries to control how parameters and predictions are inferred from the model and returned as results (rather than simply dumping raw strings from fixed queries).
(3) Value-indexed dependent types (to catch errors with vectors, matrices, and integer ranges, and help with compilation).

(1) User-Defined Functions. The Coins example shows the common pattern of a discrete distribution with a Dirichlet prior. We propose to write a function for such a pattern as follows. It explicitly returns the ret output but also implicitly returns the V output.

fun CDiscrete

N	**int**!**det**	**static input**	
R	**real**!**rnd**	**static input**	
V	**real**[N]!**rnd**	**static output**	Dirichlet[N]([**for** i $<$ N $\rightarrow$R])
ret	**mod**(N)!**rnd**	**output**	Discrete[N](V)

In a table description, **input**-attributes refer implicitly to fully observed columns in the input database. On the other hand, a function is explicitly invoked using syntax like CDiscrete(N = 2; R = 1), and the **input**-attributes N and R refer to the argument expressions, passed call-by-value.

(2) User-Defined Queries. To support both the construction of probabilistic models for inference, and the querying of results, we label each type with one of three *spaces*:

(1) **det**-space is for fully observed input data;
(2) **rnd**-space is for probabilistic models, conditioned by partially observed input data;
(3) **qry**-space is for deterministic results queried from the inferred marginal distributions of **rnd**-space variables.

We organise the three spaces via the least partial order given by **det** < **rnd** and **det** < **qry**, so as to induce a subtype relation on types. Moreover, to allow flows from **rnd**-space to **qry**-space, an operator **infer**.$D.y_i(E)$ computes the parameter y_i in **qry**-space of the marginal distribution $D(y_1,\dots,y_n)$ of an input E in **rnd**-space.

For example, here is a new model of our Coins table, using a call to CDiscrete to model the coin flips in **rnd**-space, and to implicitly define a **rnd**-space variable V for the bias of the coin. Assuming our model is conditioned by data $[1;1;0]$, the marginal distribution of V is Dirichlet[2]([2;3]) where $[2;3]$ is the counts-parameter. Hence, the call **infer**.Dirichlet[2].counts(V) yields $[2;3]$, and the query returns the mean $\frac{3}{5}$.

table Coins

Flip	**mod**(2)!**rnd**	**output**	CDiscrete(N=2, R=1.0)*(*returns Flip and Flip_V*)*
counts	**real**[2]!**qry**	**static local**	**infer**.Dirichlet[2].counts(Flip_V)
Mean	**real**!**qry**	**static output**	counts[1]/(counts[1]+counts[0])

Our reduction relation rewrites this schema to the following core form.

table Coins

R	**real**!**rnd**	**static local**	1.0
Flip_V	**real**[2]!**rnd**	**static output**	Dirichlet[2]([**for** i < 2 →R])
Flip	**mod**(2)!**rnd**	**output**	Discrete[2](V)
counts	**real**[2]!**qry**	**static local**	**infer**.Dirichlet[2].counts(Flip_V)
Mean	**real**!**qry**	**static output**	counts[1]/(counts[1]+counts[0])

(3) Simple Dependent Types. Our code has illustrated dependent types of statically-sized arrays and integer ranges: values of $T[e]$ are arrays of T of size e, while values of **mod**(e) are integers in the set $0..(e-1)$. In both cases, the size e must be a **det**-space **int**. (Hence, the dependence of types on expressions is simple, and all sizes may be resolved statically, given the sizes of tables.) The use of dependent types for arrays is standard (as in Dependent ML [30]); the main subtlety in our probabilitic setting is the need for spaces to ensure that indexes are deterministic.

Primitive distributions have dependent types:

Distributions: $D_{spc} : [x_1 : T_1,\dots,x_m : T_m](y_1 : U_1,\dots,y_n : U_n) \to T$

Discrete$_{spc}$: [N : **int**!**det**](probs : **real**!spc[N]) → **mod**(N)!**rnd**
Dirichlet$_{spc}$: [N : **int**!**det**](counts : **real**!spc[N]) → (**real**!**rnd**)[N]

User-defined functions have dependent types written as grids, such as the following type $Q_{CDiscrete}$ for CDiscrete:

N	**int!det**	**static input**
R	**real!rnd**	**static input**
V	**real[N]!rnd**	**static output**
ret	**mod(N)!rnd**	**output**

Finally, the table type for our whole model of the Coins table is the following grid. It lists the **rnd**-space variables returned by CDiscrete as well as the explicitly defined Mean. Attributes marked as **local** are private to a model or function, are identified up to alpha-equivalence, and do not appear in types. Attributes marked as **input** or **output** are binders, but are not identified up to alpha-equivalence, and are exported from tables or functions. Their names must stay fixed because of references from other tables.

V	**real[2]!rnd**	**static output**
Flip	**mod(2)!rnd**	**output**
Mean	**real!qry**	**static output**

3 Syntax of Tabular Enhanced with Functions and Queries

We describe the formal details of our revision of Tabular in this section. In the next, Section 4, we show how features such as function applications may be eliminated by reducing schemas to a core form with a direct semantics.

Column-Oriented Databases. Let t range over table names and c range over attribute names. We consider a database to be a pair $DB = (\delta_{in}, \rho_{sz})$ consisting of a record of tables $\delta_{in} = [t_i \mapsto \tau_i \ ^{i \in 1..n}]$, and a valuation $\rho_{sz} = [t_i \mapsto sz_i \ ^{i \in 1..n}]$ holding the number of rows $sz_i \in \mathbb{N}$ in each column of table t_i. Each table $\tau_i = [c_i \mapsto a_i \ ^{j \in 1..m_i}]$ is a record of attributes a_i. An *attribute* is a value V tagged with a *level* ℓ. An attribute is normally a whole *column* **inst**(V), where V is an array of length sz_i and the level **inst** is short for "instance". It may also be a single value, **static**(V), a *static attribute*. The main purpose of allowing static attributes is to return individual results (such as Mean in our Coins example) from queries.

Databases, Tables, Attributes, and Values:

$\delta_{in} ::= [t_i \mapsto \tau_i \ ^{i \in 1..n}]$	whole database
$\tau ::= [c_i \mapsto a_i \ ^{i \in 1..m}]$	table in database
$a ::= \ell(V)$	attribute value: V with level ℓ
$V ::= ? \mid s \mid [V_0, \ldots, V_{n-1}]$	nullable value
$\ell, pc ::= \textbf{static} \mid \textbf{inst}$	level (**static** $<$ **inst**)

For example, the data for our Coins example is $DB = (\delta_{in}, \rho_{sz})$ where $\delta_{in} =$ [Coins $\mapsto$ [Flip $\mapsto$ **inst**$([1;1;0])$]] and $\rho_{sz} =$ [Coins $\mapsto 3$].

In examples, we assume each table has an implicit primary key ID and that the keys are in the range $0..sz_i - 1$. A value V may contain occurrences of "?", signifying missing data; we write known(V) if V contains no occurrence of ?. Otherwise, a value may be an array, or a constant s: either a Boolean, integer, or real.

Syntax of Tabular Expressions and Schemas. An *index expression* e may be a variable x or a constant, and may occur in types (as the size of an array, for instance). Given a database $DB = (\delta_{in}, \rho_{sz})$, **sizeof**$(t)$ denotes the constant $\rho_{sz}(t)$. Attribute names c (but not table names) may occur in index expressions as variables. A *attribute type T* can be a scalar, a bounded non-negative integer or an array. Each type has an associated *space* (which is akin to an information-flow level, but independent of the notion of level in Tabular, introduced later on). (The type system is discussed in detail in Section 5.)

Index Expressions, Spaces and Dependent Types of Tabular:

$e ::= x \mid s \mid \mathbf{sizeof}(t)$	index expression
$S ::= \mathbf{bool} \mid \mathbf{int} \mid \mathbf{real}$	scalar type
$spc ::= \mathbf{det} \mid \mathbf{rnd} \mid \mathbf{qry}$	space
$T, U ::= (S\,!\,spc) \mid (\mathbf{mod}(e)\,!\,spc) \mid T[e]$	(attribute) type

$\text{space}(S\,!\,spc) \triangleq spc \quad \text{space}(T[e]) \triangleq \text{space}(T) \quad \text{space}(\mathbf{mod}(e)\,!\,spc) \triangleq spc$
$spc(T) \triangleq \text{space}(T) = spc$

We write $\mathbf{link}(t)$ as a shorthand for $\mathbf{mod}(\mathbf{sizeof}(t))$, for foreign keys to table t.

The syntax of (full) *expressions* includes index expressions, plus deterministic and random operations. We assume sets of deterministic functions g, and primitive distributions D. These have type signatures, as illustrated for Discrete and Dirichlet in Section 2. In $D[e_1,\ldots,e_m](F_1,\ldots,F_n)$, the arguments $e_1, \ldots, e_m$ index the result type, while $F_1, \ldots, F_n$ are parameters to the distribution. The operator $\mathbf{infer}.D[e_1,\ldots,e_m].y(E)$ is described intuitively in Section 2. We write $\text{fv}(T)$ and $\text{fv}(E)$ for the sets of variables occurring free in type T and expression E.

Expressions of Tabular:

$E, F ::=$	expression
e	index expression
$g(E_1,\ldots,E_n)$	deterministic primitive g
$D[e_1,\ldots,e_m](F_1,\ldots,F_n)$	random draw from distribution D
$\mathbf{if}\ E\ \mathbf{then}\ F_1\ \mathbf{else}\ F_2$	if-then-else
$[E_1,\ldots,E_n] \mid E[F]$	array literal, lookup
$[\mathbf{for}\ x < e \to F]$	for loop (scope of index x is F)
$\mathbf{infer}.D[e_1,\ldots,e_m].y(E)$	parameter y of inferred marginal of E
$E : t.c$	dereference link E to instance of c
$t.c$	dereference static attribute c of t

A Tabular schema is a relational schema with each attribute annotated not just with a type T, but also with a *level* ℓ, a *visibility viz*, and a *model expression M*.

Tabular Schemas:

$\mathbb{S} ::= [(t_1 = \mathbb{T}_1); \ldots; (t_n = \mathbb{T}_n)]$	(database) schema
$\mathbb{T} ::= [\text{col}_1; \ldots; \text{col}_n]$	table (or function)
$\text{col} ::= (c : T\ \ell\ viz\ M)$	attribute c declaration
$viz ::= \mathbf{input} \mid \mathbf{local} \mid \mathbf{output}$	visibility
$M, N ::= \varepsilon \mid E \mid M[e_{index} < e_{size}] \mid \mathbb{T}\ R$	model expression
$R ::= (c_1 = e_1, \ldots, c_n = e_n)$	function arguments

For $(c : T\ \ell\ viz\ M)$ to be well-formed, $viz = \textbf{input}$ if and only if $M = \varepsilon$. We only consider well-formed declarations. The visibility *viz* indicates whether the attribute c is given as an input, defined locally by the model expression M, or defined as an output by the model expression M. When omitted, the level of an attribute defaults to **inst**.

Functions, Models, and Model Expressions. A challenge for this paper was to find a syntax for functions that is compatible with the grid format of spreadsheets; we do so by re-interpreting the syntax $\mathbb{T}$ for tables as also the syntax of functions. A *function* is a table of the form $\mathbb{T} = [\mathsf{col}_1; \ldots ; \mathsf{col}_n; (\mathsf{ret} : T\ \textbf{output}\ E)]$. A *model* is a function where each col_i is a local or an output. A *model expression M* denotes a *model* as follows:

- An expression E denotes the model that simply returns E.
- A *function application* $\mathbb{T}\ (c_1 = e_1, \ldots, c_n = e_n)$ denotes the function $\mathbb{T}$, but with each of its inputs c_i replaced by e_i.
- An *indexed model* $M[e_{index} < e_{size}]$ denotes the model for M, but with any **rnd static** attribute c replicated e_{size} times, as an array, and with references to c replaced by the lookup $c[e_{index}]$.

Formally, functions are embedded within our syntax of function applications $\mathbb{T}\ R$. In practice, our implementation supports separate function definitions written as **fun** $f\ \mathbb{T}$, such as CDiscrete in Section 1 and CG in Section 6. A function reference (within a model expression) is written $f\ R$ to stand for $\mathbb{T}\ R$.

Indexed models appear in the original Tabular, while function applications are new.

Binders and Alpha-Equivalence. All attribute names c are considered bound by their declarations. The names of **local** attributes are identified up to alpha-equivalence. The names of **input** and **output** attributes are considered as fixed identifiers (like the fields of records) that export values from a table, and are not identified up to alpha-equivalence, because changing their names would break references to them.

Let $\mathrm{inputs}(\mathbb{T})$ be the **input** attributes of table $\mathbb{T}$, that is, the names c in $(c : T\ \ell\ \textbf{input}\ \varepsilon)$. Let $\mathrm{locals}(\mathbb{T})$ be all the **local** attributes of table $\mathbb{T}$, that is, the names c in $(c : T\ \ell\ \textbf{local}\ M)$. Let $\mathrm{outputs}(\mathbb{T})$ be all the **output** attributes of table $\mathbb{T}$, that is, the names c in $(c : T\ \ell\ \textbf{output}\ M)$ plus $\mathrm{outputs}(M)$, where the latter consists of the union of $\mathrm{outputs}(\mathbb{T}_i)$ for any applications of $\mathbb{T}_i$ within M. Let $\mathrm{dom}(\mathbb{T})$ be the union $\mathrm{inputs}(\mathbb{T}) \cup \mathrm{locals}(\mathbb{T}) \cup \mathrm{outputs}(\mathbb{T})$. Hence, the free variables $\mathrm{fv}(\mathbb{T})$ are given by:

$$\mathrm{fv}((c : T\ \ell\ viz\ M) :: \mathbb{T}') \triangleq \mathrm{fv}(T) \cup \mathrm{fv}(M) \cup (\mathrm{fv}(\mathbb{T}') \setminus (\{c\} \cup \mathrm{outputs}(M))) \qquad \mathrm{fv}([]) \triangleq \{\}$$

4 Reducing Schemas to Core Tabular

We define reduction relations that explain the meaning of function calls and indexed models by rewriting, and hence transforms any well-typed schema to a core form. The reduction semantics allows us to understand indexed models, and also function calls, within the Tabular syntax. Hence, this semantics is more direct and self-contained than the original semantics of Tabular [13], based on translating to a semantic metalanguage.

If all the attributes of a schema are simple expressions E instead of arbitrary model expressions, we say it is in *core* form:

Core Attributes, Tables, and Schemas:

$$\mathrm{Core}((c : T\ \ell\ \textbf{input}\ \varepsilon)) \quad \mathrm{Core}((c : T\ \ell\ \textbf{local}\ E)) \quad \mathrm{Core}((c : T\ \ell\ \textbf{output}\ E))$$
$$\mathrm{Core}([\mathsf{col}_1; \ldots ; \mathsf{col}_n]) \text{ if } \mathrm{Core}(\mathsf{col}_1), \ldots, \mathrm{Core}(\mathsf{col}_n)$$
$$\mathrm{Core}([t_i = \mathbb{T}_i\ ^{i \in 1..n}]) \text{ if } \mathrm{Core}(\mathbb{T}_i) \text{ for each } i \in 1..n$$

To help explain our reduction rules, consider the following function definition.

fun CG

M	**real!det static input**	
P	**real!det static input**	
Mean	**real!rnd static output**	GaussianFromMeanAndPrecision(M,P)
Prec	**real!rnd static output**	Gamma(1.0,1.0)
ret	**real!rnd output**	GaussianFromMeanAndPrecision(Mean,Prec)

The following mixture model, for a dataset consisting of durations and waiting times for Old Faithful eruptions, uses three function applications and two indexed models. Each row of the model belongs to one of two clusters, indicated by the attribute cluster; the indexed models for duration and time give different means and precisions depending on the value of cluster. Since cluster is an **output**, Tabular allows missing values in that column (and indeed they are all missing), but the **qry**-space assignment returns the most likely cluster for each row as the result of the query.

table faithful

cluster	**mod**(2)**!rnd**	**output**	CDiscrete(N=2, R=1.0)
duration	**real!rnd**	**output**	CG(M=0.0, P=1.0)[cluster < 2]
time	**real!rnd**	**output**	CG(M=60.0, P=1.0)[cluster < 2]
assignment	**mod**(2)**!qry**	**output**	ArgMax(**infer**.Discrete[2].probs(cluster))

The relation $\mathbb{T} \vdash R \rightsquigarrow_o \mathbb{T}_1$ means that $\mathbb{T}_1$ is the outcome of substituting the arguments R for the **input** attributes of the function $\mathbb{T}$, within an attribute named o. For example, for the function application in the duration attribute, we have CG $\vdash [M = 0.0, p = 1.0] \rightsquigarrow_{\text{duration}}$ CG$_1$, where CG$_1$ is as follows:

duration_Mean	**real!rnd static output**	GaussianFromMeanAndPrecision(0.0, 1.0)
duration_Prec	**real!rnd static output**	Gamma(1.0, 1.0)
duration	**real!rnd output**	GaussianFromMeanAndPrecision(duration_Mean, duration_Prec)

The inductive definition follows. Rule (Apply Input) instantiates an **input** c with an argument e; (Apply Skip) prefixes **local** and **output** attributes with o; and (Apply Ret) turns the ret attribute of the function into name o of the call-site.

Inductive Definition of Function Application: $\mathbb{T} \vdash R \rightsquigarrow_o \mathbb{T}_1$

(Apply Ret)
$$\frac{}{[(\mathsf{ret} : T\ \ell\ viz\ E)] \vdash [\,] \rightsquigarrow_o [(o : T\ \ell\ viz\ E)]}$$

(Apply Input)
$$\frac{\mathbb{T}\{^{e}/_{c}\} \vdash R \rightsquigarrow_o \mathbb{T}_1 \qquad \mathrm{dom}(\mathbb{T}) \cap \mathrm{fv}(e) = \varnothing}{(c : T\ \ell\ \textbf{input}\ \varepsilon) :: \mathbb{T} \vdash [c = e] :: R \rightsquigarrow_o \mathbb{T}_1}$$

(Apply Skip) $(viz \in \{\textbf{local}, \textbf{output}\})$
$$\frac{\mathbb{T}\{^{o\_c}/_{c}\} \vdash R \rightsquigarrow_o \mathbb{T}_1 \qquad c \notin \mathrm{fv}(R)}{(c : T\ \ell\ viz\ E) :: \mathbb{T} \vdash R \rightsquigarrow_o (o\_c : T\ \ell\ viz\ E) :: \mathbb{T}_1}$$

Next, we define $\text{index}_\sigma(\mathbb{T}, e_1, e_2)$ to be the outcome of indexing the **static rnd** or **qry** variables of a core table $\mathbb{T}$, that is, turning each declaration of such a variable into an array of size e_2, and each reference to such a variable into an array access indexed, at static level, by a local replication index i, or, at instance level, by the random indexing expression e_1 (or its mode at **qry** level). Both i and e_1 are integers bounded by e_2. Variables that require indexing are accumulated in the renaming substitution σ (which is initially empty). For instance, $\mathsf{CG}_1[\text{cluster} < 2]$ expands to $\text{index}_\varnothing(\mathsf{CG}_1, \text{cluster}, 2)$:

duration_Mean	**real**[2]!**rnd**	**static output**	[**for** i < 2 → GaussianFromMeanAndPrecision(0.0, 1.0)]
duration_Mean^	**real**!**rnd**	**local**	duration_Mean[cluster]
duration_Prec	**real**[2]!**rnd**	**static output**	[**for** i < 2 → Gamma(1.0, 1.0)]
duration_Prec^	**real**!**rnd**	**local**	duration_Prec[cluster]
duration	**real**!**rnd**	**output**	GaussianFromMeanAndPrecision(duration_Mean^, duration_Prec^)

Table Indexing: $\text{index}_\sigma(\mathbb{T}, e_1, e_2)$

$\text{index}_\sigma([], e_1, e_2) \triangleq []$
$\text{index}_\sigma((c : T\ \ell\ \textbf{input}\ \varepsilon) :: \mathbb{T}, e_1, e_2) \triangleq (c : T\ \ell\ \textbf{input}\ \varepsilon) :: (\text{index}_\sigma(\mathbb{T}, e_1, e_2))$
$\text{index}_\sigma((c : T\ \ell\ viz\ E) :: \mathbb{T}), e_1, e_2) \triangleq$
$(c : T[e_2]\ \ell\ viz\ [\textbf{for}\ i < e_2 \to \rho(E)]) :: (\hat{c} : T\ \textbf{inst local}\ c[\hat{e}_1]) :: \text{index}_{\sigma[c \mapsto \hat{c}]}(\mathbb{T}, e_1, e_2)$
if $viz \neq \textbf{input}$, $\ell = \textbf{static}$, $\neg\textbf{det}(T)$ where
$\rho = \{d \mapsto d[i] \mid d \in \text{dom}(\sigma)\}$, $i \notin \text{fv}(E) \cup \text{fv}(\sigma)$, $\hat{c} \notin \text{dom}(\mathbb{T}) \cup \text{fv}(\mathbb{T}, \sigma, c, e_1, e_2)$
and $\hat{e}_1 = e_1$ if $\textbf{rnd}(T)$, and $\hat{e}_1 = \mathsf{ArgMax}(\textbf{infer}.\mathsf{Discrete}[e_2].\mathsf{probs}(e_1))$ if $\textbf{qry}(T)$

$(c : T\ \ell\ viz\ \sigma(E)) :: \text{index}_\sigma(\mathbb{T}, e_1, e_2)$
if $viz \neq \textbf{input}$ and ($\ell = \textbf{inst}$ or $\textbf{det}(T)$)

Below, we give inductive definitions of reduction relations on schemas, tables, and model expressions. There are congruence rules, plus (RED INDEX) and (RED INDEX EXPR) for indexed models, and (RED APPL) for applications. The latter needs additional operations $\mathbb{T} \wedge \ell$ and $\mathbb{T} \wedge viz$, to adjust the model $\mathbb{T}$ of function body to the level ℓ and visibility viz of the call-site. These operators drop any **output** attributes to **local**, if the callsite is **local**, and drop any **inst**-level attributes to **static**, if the callsite is **static**.

- Consider the 2-point lattice **static** < **inst**. Let $\mathbb{T} \wedge \ell$ be the outcome of changing each $(c : T\ \ell_c\ viz\ M)$ in $\mathbb{T}$ to $(c : T\ (\ell_c \wedge \ell)\ viz\ M)$. Hence, $\mathbb{T} \wedge$ **inst** is the identity, while $\mathbb{T} \wedge$ **static** drops **inst** variables to **static** variables.
- Consider the 2-point lattice **local** < **output**. Let $\mathbb{T} \wedge viz$ be the outcome of changing each $(c : T\ \ell\ viz_c\ M)$ in $\mathbb{T}$ to $(c : T\ \ell\ (viz_c \wedge viz)\ M)$. Hence, $\mathbb{T} \wedge$ **output** is the identity, while $\mathbb{T} \wedge$ **local** drops **output** variables to **local** variables.

Reduction Relations: $\mathbb{S} \to \mathbb{S}'$, $\mathbb{T} \to \mathbb{T}'$, $M \to M'$

(RED SCHEMA LEFT)
$$\frac{\mathbb{T} \to \mathbb{T}'}{(t = \mathbb{T}) :: \mathbb{S} \to (t = \mathbb{T}') :: \mathbb{S}}$$

(RED SCHEMA RIGHT)
$$\frac{\mathbb{S} \to \mathbb{S}' \quad \mathsf{Core}(\mathbb{T})}{(t = \mathbb{T}) :: \mathbb{S} \to (t = \mathbb{T}) :: \mathbb{S}'}$$

(RED MODEL)
$$\frac{M \to M'}{(c : T\ \ell\ viz\ M) :: \mathbb{T} \to (c : T\ \ell\ viz\ M') :: \mathbb{T}}$$

(RED TABLE RIGHT)
$$\frac{\mathbb{T} \to \mathbb{T}' \quad \mathsf{Core}(\mathsf{col})}{\mathsf{col} :: \mathbb{T} \to \mathsf{col} :: \mathbb{T}'}$$

(RED INDEX INNER)

$$\frac{M \to M'}{M[e_{index} < e_{size}] \to M'[e_{index} < e_{size}]}$$

(RED INDEX)

$$\frac{\mathrm{Core}(\mathbb{T}) \quad \mathrm{fv}(e_{index}, e_{size}) \cap (\mathrm{dom}(\mathbb{T})) = \varnothing}{(\mathbb{T}\ R)[e_{index} < e_{size}] \to (\mathrm{index}_{\varnothing}(\mathbb{T}, e_{index}, e_{size}))\ R}$$

(RED INDEX EXPR)

$$\frac{}{E[e_{index} < e_{size}] \to E}$$

(RED APPL) (for Core($\mathbb{T}$))

$$\frac{((\mathbb{T} \wedge \ell) \wedge viz) \vdash R \rightsquigarrow_o \mathbb{T}_1 \quad (\mathrm{locals}(\mathbb{T}_1) \cup \mathrm{inputs}(\mathbb{T}_1)) \cap (\mathrm{fv}(\mathbb{T}') \cup \mathrm{dom}(\mathbb{T}')) = \varnothing}{(o : T'\ \ell\ viz\ (\mathbb{T}\ R)) :: \mathbb{T}' \to \mathbb{T}_1 @ \mathbb{T}'}$$

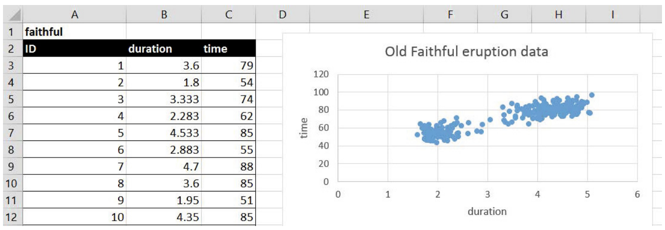
Tables in core form have no reductions. Moreover, the reduction relation is deterministic (we include the Core($\mathbb{T}$) condition on the rules (RED SCHEMA RIGHT), (RED TABLE RIGHT), and (RED INDEX) to fix a particular reduction strategy).

By using the above rules to expand out the three function calls and the two model expressions in the Old Faithful example, we obtain the core model below:

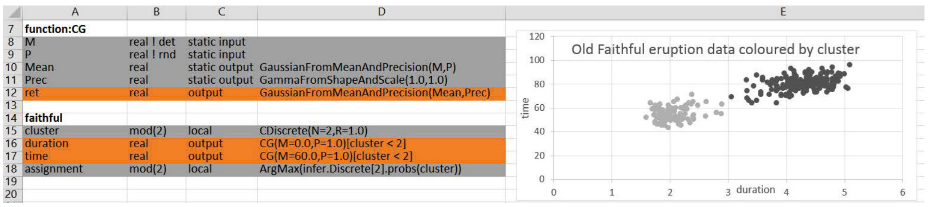
table faithful

cluster_V	**real**[2]!**rnd**	**static output**	Dirichlet[2]([**for** i < 2 → 1.0])
cluster	**mod**(2)!**rnd**	**output**	Discrete[2](cluster_V)
duration_Mean	**real**[2]!**rnd**	**static output**	[**for** i < 2 → GaussianFromMeanAndPrecision(0.0, 1.0)]
duration_Mean^	**real**!**rnd**	**local**	duration_Mean[cluster]
duration_Prec	**real**[2]!**rnd**	**static output**	[**for** i < 2 → Gamma(1.0, 1.0)]
duration_Prec^	**real**!**rnd**	**local**	duration_Prec[cluster]
duration	**real**!**rnd**	**output**	GaussianFromMeanAndPrecision(duration_Mean^, duration_Prec^)
time_Mean	**real**[2]!**rnd**	**static output**	[**for** i < 2 → GaussianFromMeanAndPrecision(60.0, 1.0)]
time_Mean^	**real**!**rnd**	**local**	time_Mean[cluster]
time_Prec	**real**[2]!**rnd**	**static output**	[**for** i < 2 → Gamma(1.0, 1.0)]
time_Prec^	**real**!**rnd**	**local**	time_Prec[cluster]
time	**real**!**rnd**	**output**	GaussianFromMeanAndPrecision(time_Mean^, time_Prec^)
assignment	**mod**(2)!**qry**	**output**	ArgMax(**infer**.Discrete[2].probs(cluster))

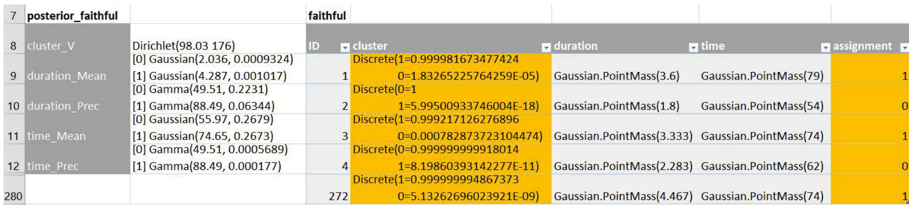
Moreover, here are screen shots (best viewed in colour) of the data, model and inference results in Excel.



	A	B	C
1	**faithful**		
2	**ID**	**duration**	**time**
3	1	3.6	79
4	2	1.8	54
5	3	3.333	74
6	4	2.283	62
7	5	4.533	85
8	6	2.883	55
9	7	4.7	88
10	8	3.6	85
11	9	1.95	51
12	10	4.35	85



	A	B	C	D
7	**function:CG**			
8	M	real ! det	static input	
9	P	real ! rnd	static input	
10	Mean	real	static output	GaussianFromMeanAndPrecision(M,P)
11	Prec	real	static output	GammaFromShapeAndScale(1.0,1.0)
12	ret	real	output	GaussianFromMeanAndPrecision(Mean,Prec)
13				
14	**faithful**			
15	cluster	mod(2)	local	CDiscrete(N=2,R=1.0)
16	duration	real	output	CG(M=0.0,P=1.0)[cluster < 2]
17	time	real	output	CG(M=60.0,P=1.0)[cluster < 2]
18	assignment	mod(2)	local	ArgMax(infer.Discrete[2].probs(cluster))
19				
20				



7	**posterior_faithful**		**faithful**				
8	cluster_V	Dirichlet(98.03 176)	ID	cluster	duration	time	assignment
9	duration_Mean	[0] Gaussian(2.036, 0.0009324) [1] Gaussian(4.287, 0.001017)	1	Discrete(1=0.9999816734774240=1.83265225764259E-05)	Gaussian.PointMass(3.6)	Gaussian.PointMass(79)	1
10	duration_Prec	[0] Gamma(49.51, 0.2231) [1] Gamma(88.49, 0.06344)	2	Discrete(0=1 1=5.99500933746004E-18)	Gaussian.PointMass(1.8)	Gaussian.PointMass(54)	0
11	time_Mean	[0] Gaussian(55.97, 0.2679) [1] Gaussian(74.65, 0.2673)	3	Discrete(1=0.999217126276896 0=0.000782873723104474)	Gaussian.PointMass(3.333)	Gaussian.PointMass(74)	1
12	time_Prec	[0] Gamma(49.51, 0.0005689) [1] Gamma(88.49, 0.000177)	4	Discrete(0=0.999999999918014 1=8.19860393142277E-11)	Gaussian.PointMass(2.283)	Gaussian.PointMass(62)	0
280			272	Discrete(1=0.999999994867373 0=5.13262696023921E-09)	Gaussian.PointMass(4.467)	Gaussian.PointMass(74)	1

5 Dependent Type System and Semantics

5.1 Dependent Type System

The expressions of Tabular are based on the probabilistic language Fun [2]. We significantly extend Fun by augmenting its types with the three spaces described in Section 1, adding value-indexed dependent types including statically bounded integers and sized arrays, and additional expressions including an operator for inference and operations for referencing attributes of tables and their instances.

We use *table types* Q both for functions and for concrete tables. When used to type a function Q must satisfy the predicate $\mathsf{\mathbf{fun}}(Q)$, which requires it to use the distinguished name ret for the explicit result of the function (its final output). When used to type a concrete table Q must satisfy the predicate $\mathsf{\mathbf{table}}(Q)$, which ensures that types do not depend on the contents of any input table t (except for the sizes of tables). We only need $\mathsf{\mathbf{table}}(Q)$ to define a conformance relation on databases and schema types.

Table and Schema Types:

$Q ::= [(c_i : T_i\ \ell_i\ viz_i)\ ^{i \in 1..n}]$	table type (c_i distinct, $viz_i \neq \mathsf{\mathbf{local}}$)
$Sty ::= [(t_i : Q_i)\ ^{i \in 1..n}]$	schema type (t_i distinct)

$\mathsf{\mathbf{fun}}(Q)$ iff $viz_n = \mathsf{\mathbf{output}}$ and $c_n = \mathsf{ret}$.
$\mathsf{\mathbf{model}}(Q)$ iff $\mathsf{\mathbf{fun}}(Q)$ and each $viz_i = \mathsf{\mathbf{output}}$.
$\mathsf{\mathbf{table}}(Q)$ iff for each $i \in 1..n$, $\ell_i = \mathsf{\mathbf{static}} \Rightarrow \mathsf{\mathbf{rnd}}(T_i) \vee \mathsf{\mathbf{qry}}(T_i)$.

Tabular typing environments Γ are ordered maps associating variables with their declared level and type, and table identifiers with their inferred table types. The typing rules will prevent expressions typed at level **static** from referencing **inst** level variables.

Tabular Typing Environments:

$\Gamma ::= \varnothing \mid (\Gamma, x :^{\ell} T) \mid (\Gamma, t : Q)$	environment
$\gamma([(c_i : T_i\ \ell_i\ viz_i)\ ^{i \in 1..n}]) \triangleq c_i :^{\ell_i} T_i\ ^{i \in 1..n}$	Q as an environment

Next, we present the judgments and rules of the type system.

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 level pc, index expression e has type T
$\Gamma \vdash Q$	in Γ, table type Q is well-formed
$\Gamma \vdash Sty$	in Γ, schema type Sty is well-formed
$\Gamma \vdash T <: U$	in Γ, T is a subtype of U
$\Gamma \vdash^{pc} E : T$ path	in Γ at level pc, expression E is a path
$\Gamma \vdash^{pc} E : T$	in Γ at level pc, expression E has type T
$\Gamma \vdash^{pc}_{o} R : Q \to Q'$	R sends function type Q to model type Q' in column o
$\Gamma \vdash^{pc}_{o} M : Q$	model expression M has model type Q
$\Gamma \vdash^{pc} \mathbb{T} : Q$	table $\mathbb{T}$ has type Q
$\Gamma \vdash \mathbb{S} : Sty$	schema $\mathbb{S}$ has type Sty

The formation rules for types and environments depend mutually on the typing rules for index expressions. Only index expressions that are **det**-space and **static**-level may occur in types. We write ty(s) for the scalar type S of the scalar s.

Rules for Types, Environments, and Index Expressions: $\Gamma \vdash \diamond \quad \Gamma \vdash T \quad \Gamma \vdash^{pc} e : T$

(ENV EMPTY)
$$\frac{}{\varnothing \vdash \diamond}$$

(ENV VAR)
$$\frac{\Gamma \vdash T \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma, x :^{pc} T \vdash \diamond}$$

(ENV TABLE) ($\mathbf{table}(Q)$)
$$\frac{\Gamma \vdash Q \quad t \notin \mathrm{dom}(\Gamma)}{\Gamma, t : Q \vdash \diamond}$$

(TYPE SCALAR)
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash S \,!\, spc}$$

(TYPE RANGE)
$$\frac{\Gamma \vdash^{\mathbf{static}} e : \mathbf{int} \,!\, \mathbf{det}}{\Gamma \vdash \mathbf{mod}(e) \,!\, spc}$$

(TYPE ARRAY)
$$\frac{\Gamma \vdash T \quad \Gamma \vdash^{\mathbf{static}} e : \mathbf{int} \,!\, \mathbf{det}}{\Gamma \vdash T[e]}$$

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

(INDEX SCALAR)
$$\frac{\Gamma \vdash \diamond \quad S = \mathrm{ty}(s)}{\Gamma \vdash^{pc} s : S \,!\, \mathbf{det}}$$

(INDEX MOD)
$$\frac{\Gamma \vdash \diamond \quad 0 \leq n < m}{\Gamma \vdash^{pc} n : \mathbf{mod}(m) \,!\, \mathbf{det}}$$

(INDEX SIZEOF)
$$\frac{\Gamma \vdash \diamond \quad \Gamma = \Gamma', t : Q, \Gamma''}{\Gamma \vdash^{pc} \mathbf{sizeof}(t) : \mathbf{int} \,!\, \mathbf{det}}$$

Formation Rules for Table and Schema Types: $\Gamma \vdash Q \quad \Gamma \vdash Sty$

(TABLE TYPE [])
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash [] : []}$$

(TABLE TYPE INPUT)
$$\frac{\Gamma \vdash T \quad \Gamma, c :^{\ell} T \vdash Q}{\Gamma \vdash (c : T\ \ell\ \mathbf{input}) :: Q}$$

(TABLE TYPE OUTPUT)
$$\frac{\Gamma \vdash T \quad \Gamma, c :^{\ell} T \vdash Q}{\Gamma \vdash (c : T\ \ell\ \mathbf{output}) :: Q}$$

(SCHEMA TYPE [])
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash [] : []}$$

(SCHEMA TYPE TABLE)
$$\frac{\Gamma \vdash Q \quad \mathbf{table}(Q) \quad \Gamma, t : Q \vdash Sty}{\Gamma \vdash (t : Q) :: Sty}$$

Subtyping allows **det**-space data to be used as **rnd**-space or **qry**-space data. The preorder $\leq$ on spaces is the least reflexive relation to satisfy $\mathbf{det} \leq \mathbf{rnd}$ and $\mathbf{det} \leq \mathbf{qry}$. The default space is **det**, so when we write S or $\mathbf{mod}(e)$ as a type, we mean $S \,!\, \mathbf{det}$ or $\mathbf{mod}(e) \,!\, \mathbf{det}$. We define a commutative partial operation $spc \vee spc'$, and lift this operation to types $T \vee spc$ to weaken the space of a type.

Least upper bound: $spc \vee spc'$ **(if** $spc \leq spc'$ **or** $spc' \leq spc$**)**

$spc \vee spc = spc \qquad \mathbf{det} \vee \mathbf{rnd} = \mathbf{rnd} \qquad \mathbf{det} \vee \mathbf{qry} = \mathbf{qry}$

(The combination $\mathbf{rnd} \vee \mathbf{qry}$ is intentionally not defined.)

Operations on Types and Spaces: $T \vee spc$

$(S\,!\,spc) \vee spc' \triangleq S\,!\,(spc \vee spc') \qquad T[e] \vee spc \triangleq (T \vee spc)[e]$

$(\mathbf{mod}(e)\,!\,spc) \vee spc' \triangleq \mathbf{mod}(e)\,!\,(spc \vee spc')$

Given these definitions, we present the rules of subtyping and of typing expressions.

Rules of Subtyping: $\Gamma \vdash T <: U$

(SUB SCALAR)
$$\frac{\Gamma \vdash \diamond \quad spc_1 \leq spc_2}{\Gamma \vdash S\,!\,spc_1 <: S\,!\,spc_2}$$

(SUB MOD)
$$\frac{\Gamma \vdash^{\mathsf{static}} e : \mathbf{int}\,!\,\mathbf{det} \quad spc_1 \leq spc_2}{\Gamma \vdash \mathbf{mod}(e)\,!\,spc_1 <: \mathbf{mod}(e)\,!\,spc_2}$$

(SUB ARRAY)
$$\frac{\Gamma \vdash T <: U \qquad \Gamma \vdash^{\mathsf{static}} e : \mathbf{int}\,!\,\mathbf{det}}{\Gamma \vdash T[e] <: U[e]}$$

The table below presents the typing rules for Tabular expressions, most of which are standard modulo the operations on spaces. For instance, in (DEREF INST), the type of the indexed column needs to be joined with the space of the index, because, for instance, an expression returning a deterministic value at a random index is random. Similarly, an expression returning an element of a deterministic array at a random index is random, hence the join in (INDEX).

Since deterministic parameters of random primitives can occur in the types of random arguments and the return type, they have to be substituted out in the (RANDOM) and (INFER) rules.

(Selected) Typing Rules for Expressions: $\Gamma \vdash^{pc} E : T$ path**,** $\Gamma \vdash^{pc} E : T$

(VARIABLE PATH)
$$\frac{\Gamma \vdash^{pc} x : T}{\Gamma \vdash^{pc} x : T \text{ path}}$$

(INDEXED PATH)
$$\frac{\Gamma \vdash^{pc} p_1 : T[e] \text{ path} \quad \Gamma \vdash^{pc} p_2 : \mathbf{mod}(e)!\text{det path}}{\Gamma \vdash^{pc} p_1[p_2] : T \text{ path}}$$

(SUBSUM)
$$\frac{\Gamma \vdash^{pc} E : T \quad \Gamma \vdash T <: U}{\Gamma \vdash^{pc} E : U}$$

(INDEX EXPRESSION)
$$\frac{\Gamma \vdash^{pc} e : T \qquad (e \text{ is an index expression})}{\Gamma \vdash^{pc} e : T \qquad (e \text{ seen as an expression})}$$

(DEREF STATIC)
$$\frac{\Gamma = \Gamma', t : Q, \Gamma'' \qquad Q = Q' @ [(c : T \ \mathbf{static}\ viz)] @ Q''}{\Gamma \vdash^{pc} t.c : T}$$

(DEREF INST)
$$\frac{\Gamma \vdash^{pc} E : \mathbf{link}(t)\,!\,spc \qquad \Gamma = \Gamma', t : Q, \Gamma'' \quad Q = Q' @ [(c : T \ \mathbf{inst}\ viz)] @ Q''}{\Gamma \vdash^{pc} E : t.c : T \vee spc}$$

(RANDOM) (where $\sigma(U) \triangleq U\{^{e_1}/_{x_1}\}\ldots\{^{e_m}/_{x_m}\}$)
$$\frac{\begin{array}{l} D_{\mathsf{rnd}} : [x_1 : T_1, \ldots, x_m : T_m](y_1 : U_1, \ldots, y_n : U_n) \to T \\ \Gamma \vdash^{\mathsf{static}} e_i : T_i \quad \forall i \in 1..m \quad \Gamma \vdash^{pc} F_j : \sigma(U_j) \quad \forall j \in 1..n \quad \Gamma \vdash \diamond \\ \{x_1, \ldots, x_m\} \cap (\bigcup_i \mathrm{fv}(e_i)) = \varnothing \quad x_i \neq x_j \text{ for } i \neq j \end{array}}{\Gamma \vdash^{pc} D[e_1, \ldots, e_m](F_1, \ldots, F_n) : \sigma(T)}$$

(ITER) (where $x \notin \mathrm{fv}(T)$)

$$\frac{\Gamma \vdash^{\mathsf{static}} e : \mathbf{int}\,!\,\mathbf{det} \qquad \Gamma, x :^{pc} (\mathbf{mod}(e)\,!\,\mathbf{det}) \vdash^{pc} F : T}{\Gamma \vdash^{pc} [\mathbf{for}\ x < e \to F] : T[e]}$$

(INDEX)

$$\frac{\mathrm{space}(T) \leq spc \qquad \Gamma \vdash^{pc} E : T[e] \quad \Gamma \vdash^{pc} F : \mathbf{mod}(e)\,!\,spc}{\Gamma \vdash^{pc} E[F] : T \vee spc}$$

(INFER) (where $\sigma(U) \triangleq U\{^{e_1}/_{x_1}\}\ldots\{^{e_m}/_{x_m}\}$)

$$\frac{\begin{array}{l} D_{\mathsf{qry}} : [x_1 : T_1, \ldots, x_m : T_m](y_1 : U_1, \ldots, y_n : U_n) \to T \\ \Gamma \vdash^{\mathsf{static}} e_i : T_i \quad \forall i \in 1..m \quad \Gamma \vdash^{pc} E : \sigma(T)\ \mathsf{path} \quad j \in 1..n \\ \{x_1, \ldots, x_m\} \cap (\bigcup_i \mathrm{fv}(e_i)) = \varnothing \quad x_i \neq x_j \text{ for } i \neq j \end{array}}{\Gamma \vdash^{pc} \mathbf{infer}.D[e_1, \ldots, e_m].y_j(E) : \sigma(U_j)}$$

For an example of (INFER), recall the expression **infer**.Dirichlet[2].counts(V) from Section 1. Here $m = n = 1$, $y_1 =$ counts and $U_1 = \mathbf{real}[\mathsf{N}]!\mathbf{qry}$ and $\sigma = \{^2/_{\mathsf{N}}\}$ and the result type is $\sigma(U_1) = \mathbf{real}[2]!\mathbf{qry}$.

Below are the typing rules for function arguments. In (ARG INPUT), the level $\ell \wedge pc$ at which the argument needs to be checked is bounded both by the level pc of the function aplication and the level ℓ of the given column of the function. In (ARG OUTPUT), the level $\ell \wedge pc$ of the output column of the reduced application is bounded by the level pc at which the function was applied as well as the level ℓ of the given column.

Typing Rules for Arguments: $\Gamma \vdash^{pc}_{o} R : Q \to Q'$

(ARG INPUT)
$$\frac{\Gamma \vdash^{\ell \wedge pc} e : T \quad \Gamma \vdash^{pc}_{o} R : Q\{^{e}/_{c}\} \to Q'}{\Gamma \vdash^{pc}_{o} ((c = e) :: R) : ((c : T\ \ell\ \mathbf{input}) :: Q) \to Q'}$$

(ARG OUTPUT)
$$\frac{\Gamma \vdash T \qquad \Gamma \vdash^{pc}_{o} R : Q\{^{o\_c}/_{c}\} \to Q' \qquad c \neq \mathsf{ret}}{\Gamma \vdash^{pc}_{o} R : ((c : T\ \ell\ \mathbf{output}) :: Q) \to ((o\_c : T\ (\ell \wedge pc)\ \mathbf{output}) :: Q')}$$

(ARG RET)
$$\frac{\Gamma \vdash T}{\Gamma \vdash^{pc}_{o} R : (\mathsf{ret} : T\ \ell\ \mathbf{output}) \to (\mathsf{ret} : T\ (\ell \wedge pc)\ \mathbf{output})}$$

For example, if $Q_{CDiscrete}$ is the function type of CDiscrete from Section 1 we can derive $\mathsf{b} :^{\mathsf{static}} \mathbf{real}!\mathbf{rnd} \vdash^{\mathsf{inst}}_{\mathsf{Flip}} (\mathsf{N} = 2, \mathsf{alpha} = \mathsf{b}) : Q_{CDiscrete} \to Q'$ where Q', shown in the grid below, represents the outputs of the function call. Since the inputs N and alpha of CDiscrete are both **static**, arguments 2 and b are typed at level **static** $\wedge$ **inst** $=$ **static**.

Flip_V	**real**[2]!**rnd**	**static output**
ret	**mod**(2)!**rnd**	**output**

Next, we have rules for assigning a model type Q to a model expression M. (MODEL INDEXED) needs the following operation $Q[e]$ to capture the static effect of indexing:

Indexing a Table Type: $Q[e]$

$\varnothing[e] \triangleq \varnothing$

$((c : T\ \mathbf{inst}\ viz) :: Q)[e] \triangleq (c : T\ \mathbf{inst}\ viz) :: (Q[e])$

$((c : T\ \mathbf{static}\ viz) :: Q)[e] \triangleq (c : T\ \mathbf{static}\ viz) :: (Q[e])$ if $viz = \mathbf{input}$ or $\mathbf{det}(T)$

$((c : T\ \mathbf{static}\ viz) :: Q)[e] \triangleq (c : T[e]\ \mathbf{static}\ viz) :: (Q[e])$ if $viz \neq \mathbf{input}$ and $\neg\mathbf{det}(T)$

The vectorized c cannot appear in Q when $\mathbf{rnd}(T)$, so $Q[e]$ remains well-formed.

Typing Rules for Model Expressions: $\Gamma \vdash_o^{pc} M : Q$

(MODEL EXPRESSION)
$$\frac{\Gamma \vdash^{pc} E : T}{\Gamma \vdash_o^{pc} E : [(\mathsf{ret} : T\ pc\ \mathbf{output})]}$$

(MODEL APPL)
$$\frac{\Gamma \vdash^{pc} \mathbb{T} : Q \quad \mathbf{fun}(Q) \quad \Gamma \vdash_o^{pc} R : Q \to Q'}{\Gamma \vdash_o^{pc} \mathbb{T}\, R : Q'}$$

(MODEL INDEXED)
$$\frac{\Gamma \vdash_o^{pc} M : Q \quad \mathrm{dom}(Q) \cap \mathrm{fv}(e_{size}) = \varnothing \quad \Gamma \vdash^{pc} e_{index} : \mathbf{mod}(e_{size})\ !\ \mathbf{rnd}\ \mathsf{path}}{\Gamma \vdash_o^{pc} M[e_{index} < e_{size}] : Q[e_{size}]}$$

Finally, we complete the system with rules for tables and schemas.

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

(TABLE [])
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash^{pc} [] : []}$$

(TABLE INPUT)
$$\frac{\Gamma, c :^{\ell \wedge pc} T \vdash^{pc} \mathbb{T} : Q}{\Gamma \vdash^{pc} (c : T\ \ell\ \mathbf{input}\ \varepsilon) :: \mathbb{T} : (c : T\ (\ell \wedge pc)\ \mathbf{input}) :: Q}$$

(TABLE OUTPUT)
$$\frac{\Gamma \vdash_c^{\ell \wedge pc} M : Q_c @ [(\mathsf{ret} : T\ (\ell \wedge pc)\ \mathbf{output})] \quad \Gamma, \gamma(Q_c), c :^{\ell \wedge pc} T \vdash^{pc} \mathbb{T} : Q}{\Gamma \vdash^{pc} (c : T\ \ell\ \mathbf{output}\ M) :: \mathbb{T} : Q_c @ ((c : T\ (\ell \wedge pc)\ \mathbf{output}) :: Q)}$$

(TABLE LOCAL) (where $(\mathrm{dom}(Q_c) \cup \{c\}) \cap \mathrm{fv}(Q) = \varnothing$)
$$\frac{\Gamma \vdash_c^{\ell \wedge pc} M : Q_c @ [(\mathsf{ret} : T\ (\ell \wedge pc)\ \mathbf{output})] \quad \Gamma, \gamma(Q_c), c :^{\ell \wedge pc} T \vdash^{pc} \mathbb{T} : Q}{\Gamma \vdash^{pc} (c : T\ \ell\ \mathbf{local}\ M) :: \mathbb{T} : Q}$$

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

(SCHEMA [])
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash [] : []}$$

(SCHEMA TABLE)
$$\frac{\Gamma \vdash^{\mathsf{inst}} \mathbb{T} : Q \quad \mathbf{table}(Q) \quad \Gamma, t : Q \vdash \mathbb{S} : Sty}{\Gamma \vdash (t = \mathbb{T}) :: \mathbb{S} : (t : Q) :: Sty}$$

5.2 Reduction to Core Tabular

Proposition 1 (Preservation).

(1) *If* $\Gamma \vdash \mathbb{S} : Sty$ *and* $\mathbb{S} \to \mathbb{S}'$ *then* $\Gamma \vdash \mathbb{S}' : Sty$.
(2) *If* $\Gamma \vdash^{pc} \mathbb{T} : Q$ *and* $\mathbb{T} \to \mathbb{T}'$ *then* $\Gamma \vdash^{pc} \mathbb{T}' : Q$.
(3) *If* $\Gamma \vdash^{pc} M : Q$ *and* $M \to M'$ *then* $\Gamma \vdash^{pc} M' : Q$.

Proposition 2 (Progress). *If* $\Gamma \vdash^{pc} \mathbb{S} : Q$ *either* $\mathrm{Core}(\mathbb{S})$ *or there is* $\mathbb{S}'$ *such that* $\mathbb{S} \to \mathbb{S}'$.

Proposition 3 (Termination). *No infinite chain* $\mathbb{S}_0 \to \mathbb{S}_1 \to \ldots$ *exists.*

Algorithm 1. Reducing to Core Schema: $\mathsf{CoreSchema}(\mathbb{S})$

(1) Compute $\mathbb{S}'$ such that $\mathbb{S} \to^* \mathbb{S}'$ and $\mathsf{Core}(\mathbb{S}')$.
(2) Output $\mathbb{S}'$.

As a corollary of our three propositions, we obtain:

Theorem 1. *If* $\varnothing \vdash \mathbb{S} : Sty$ *then* $\mathsf{CoreSchema}(\mathbb{S})$ *terminates with a unique schema* $\mathbb{S}'$ *such that* $\mathbb{S} \to^* \mathbb{S}'$ *and* $\mathrm{Core}(\mathbb{S}')$ *and* $\varnothing \vdash \mathbb{S}' : Sty$*.*

5.3 Semantics of Core Tabular (Sketch)

Following [2], we define a semantics based on measure theory for **det** and **rnd**-level attributes, plus a set of evaluation rules for **qry**-level variables. For the sake of brevity, we omit the precise definitions here, and instead sketch the semantics and state the key theoretical result, illustrating it by example. For full details, see [15].

The denotational semantics of a schema $\mathbb{S}$ with respect to an input database δ_{in} is a measure μ defined on the measurable space corresponding to this schema. In order to evaluate the queries in the schema, we need to compute marginal measures for all (non-**qry**) attributes of all tables.

More precisely, our semantics for Tabular factors into an idealised, probabilistic denotational semantics (abstracting the details of approximate inference algorithms such as Infer.NET and other potential implementations) and a mostly conventional operational semantics.

The denotational semantics interprets well-typed schema as inductively defined measurable spaces, $\boldsymbol{T}[\![\mathbb{S}]\!]^{\rho_{sz}}$, and defines a (mathematical) function interpreting well typed schemas $\boldsymbol{P}[\![\mathbb{S}]\!]^{\delta_{in}}_{(\tau,\delta)} \in \boldsymbol{T}[\![\mathbb{S}]\!]^{\rho_{sz}}$ as sub-probability measures describing the joint distribution μ of random variables given the observed input database δ_{in}.

The relation $\delta \vdash_\sigma \mathbb{S} \Downarrow \delta_{out}$ of our operational semantics takes as input a nested map σ of marginal measures for each column in the database, and the current operational environment δ (a nested map from **qry** and **det** attributes to values), and a schema. It returns an (output) database value δ_{out}: a nested map that assigns values to each non-**rnd** attribute of the schema.

Algorithm 2. Query Semantics of Core Schema: $\mathsf{CoreQuery}(\mathbb{S}, DB)$

```
(1) Assume core($\mathbb{S}$) and $DB = (\delta_{in}, \rho_{sz})$.
(2) Let $\mu \triangleq \boldsymbol{P}[\![\mathbb{S}]\!]^{\delta_{in}}_{[]}$ (that is, the joint distribution over all rnd-variables).
(3) Let $\sigma = \mathsf{marginalize}(\mathbb{S}, \rho_{sz}, \mu)$.
(4) Return $(\delta_{out}, \rho_{sz})$ such that $\varnothing \vdash_\sigma \mathbb{S} \Downarrow \delta_{out}$.
```

Theorem 2 below states that, given a well-typed schema and conforming database, the composition of the denotational semantics and the deterministic evaluation relation yields a well-typed output database (with the same dimensions). The notation $DB \models^{\mathsf{in}} Sty$ means that the database DB is a well-formed input to Sty; dually, $DB \models^{\mathsf{out}} Sty$ means that the database DB is a well-formed output of Sty.

Theorem 2. *Suppose* $\mathrm{Core}(\mathbb{S})$ *and* $\varnothing \vdash \mathbb{S} : Sty$ *and* $DB = (\delta_{in}, \rho_{sz})$ *and* $DB \models^{\mathsf{in}} Sty$*. Then algorithm* $\mathsf{CoreQuery}(\mathbb{S}, DB)$ *returns* $DB' = (\delta_{out}, \rho_{sz})$ *such that* $DB' \models^{\mathsf{out}} Sty$*.*

To illustrate, consider the Old Faithful schema shown in Section 4, together with an input database $(\delta_{in}, [\mathsf{faithful} \mapsto 272])$ with 272 rows (say) such as the following:

$$\delta_{in} = [\,\mathsf{faithful} \mapsto [\,\mathsf{duration} \mapsto \mathbf{inst}([1.9;4.0;4.9;\ldots)];\ \ \mathsf{time} \mapsto \mathbf{inst}([50;75;80;\ldots])\,]\,]$$

The output of the marginalisation algorithm for the only table in this schema is:

$$\begin{aligned}
[\ &\mathsf{cluster\_V} \mapsto \textbf{static}(\mu_{\text{Dirichlet}[2](1,1)});\\
&\mathsf{cluster} \mapsto \textbf{inst}([\mu_{20};\mu_{21};\mu_{22};\ldots]);\\
&\mathsf{duration\_Mean} \mapsto \textbf{static}([\mu_{\text{Gaussian}(0,1)},\mu_{\text{Gaussian}(0,1)}]);\\
&\mathsf{duration\_Prec} \mapsto \textbf{static}([\mu_{\text{Gamma}(1,1)},\mu_{\text{Gamma}(1,1)}]);\\
&\mathsf{duration} \mapsto \textbf{inst}([\mu_{50};\mu_{51};\mu_{52};\ldots]);\\
&\mathsf{time\_Mean} \mapsto \textbf{static}([\mu_{\text{Gaussian}(60,1)},\mu_{\text{Gaussian}(60,1)}]);\\
&\mathsf{time\_Prec} \mapsto \textbf{static}([\mu_{\text{Gamma}(1,1)},\mu_{\text{Gamma}(1,1)}]);\\
&\mathsf{time} \mapsto \textbf{inst}([\mu_{80};\mu_{81};\mu_{82};\ldots])\ ]
\end{aligned}$$

The output database is $(\delta_{out}, [\mathsf{faithful} \mapsto 272])$ where δ_{out} contains those entries from such environments which correspond to non-random attributes (that is, are not measures). In our example, it is of the form:

$$\delta_{out} = [\ \mathsf{faithful} \mapsto [\ \mathsf{assignment} \mapsto \textbf{inst}([0;1;1;\ldots])\ ]\ ]$$

In this example, all of the **inst** arrays are of length 272.

6 Examples of Bayesian Decision Analysis in Tabular

To illustrate the value of query-space computations, we illustrate how they express a range of decision problems. Decisions such as these cannot be expressed in the original form of Tabular. Other probabilistic programming languages have built in constructs for decision-making, whereas Tabular does so using ideas of information flow.

We describe how three example decision problems are written as Tabular queries. The result of Bayesian inference is the posterior belief over quantities of interest, including model parameters such as the **rnd**-space variable V in our coins example. These inferences reflect a change of belief in light of data, but they are not sufficient for making decisions, which requires optimization under uncertainty.

In Bayesian Decision Analysis, the decision making process is based on statistical inference followed by maximization of expected utility of the outcome. Following Gelman et al. [8], Bayesian Decision Analysis can be described as follows:

(1) Enumerate sets D and X of all possible decision options $d \in D$ and outcomes $x \in X$.
(2) Determine the probability distribution over outcomes $x \in X$ conditional on each decision option $d \in D$.
(3) Define a utility function $U : X \to \mathbb{R}$ to value each outcome.
(4) Calculate the expected utility $E[U(x)|d]$ as a function of decision option d and make the decision with the highest expected utility.

(1) Optimal Betting Decisions. Consider a situation in which to decide whether or not to accept a given sports bet based on the TrueSkill model for skill estimation [16]. The following code shows the schema $\mathbb{S}_{TrueSkill}$. Following [13], the tables Players and Matches generate **rnd**-space variables for the results of matches between players, by comparison of their per-match performances, modelled as noisy per-player skills.

table Players

Skill	**real**!**rnd**	**output**	Gaussian(25.0,100.0)

table Matches

Player1	**link**(Players)!**det**	**input**	
Player2	**link**(Players)!**det**	**input**	
Perf1	**real**!**rnd**	**output**	Gaussian(Player1.Skill,100.0)
Perf2	**real**!**rnd**	**output**	Gaussian(Player2.Skill,100.0)
Win1	**bool**!**rnd**	**output**	Perf1 > Perf2

table Bets

Match	**link**(Matches)!**det**	**input**	
Odds1	**real**!**det**	**input**	
Win1	**bool**!**rnd**	**output**	Match.Win1
p	**real**!**qry**	**output**	**infer**.Bernoulli[].Bias(Win1)
U	**real**[3]!**det**	**output**	[0.0;−1.0;Odds1 ∗ 1.0]
EU	**real**[2]!**qry**	**output**	[U[0];((1.0 − p)∗ U[1])+ (p ∗ U[2])]
PlaceBet1	**mod**(2)!**qry**	**output**	ArgMax(EU)

Table Bets represents the decision theoretic part of the code and refers to Matches together with the odds Odds1 offered for a bet on player 1 winning. Here, the two decision options in $D = \{0,1\}$ are to take the bet (PlaceBet1 $= 1$) or not (PlaceBet1 $= 0$), and the three possible outcomes in $X = \{0,1,2\}$ are abstain $= 0$, loss $= 1$ or win $= 2$. The optimal decision depends on the odds: a risky bet may be worth taking if the odds are good. The utility function U is given by money won for a fixed bet size of, say, \$1, so $U(\text{abstain}) = 0.0$, $U(\text{loss}) = -\$1.0$, and $U(\text{win}) = \text{Odds1} * \$1.0$. Variable p is obtained from **qry** expression **infer**.Bernoulli.Bias(Win1) and represents the inferred probability of a positive bet outcome. The **qry** variable PlaceBet1 is 1 if the expected utility $EU[1] = (1-p)\cdot(-\$1.0) + p \cdot \text{Odds1} \cdot \$1.0$ is greater than $EU[0] = 0.0$, that is, betting is better than not betting. The ArgMax operator simply returns the first index (of type **mod**(n)) of the maximum value in its array argument (of type **real**$[n]$). It returns the decision delivering the maximum expected utility.

(2) Classes with Asymmetric Misclassification Costs. Consider the task of n-ary classification with class-specific misclassification costs. We proceed by defining the schema for a Naive Bayes classifier (see, for instance, Duda and Hart [7]), in terms of the function CG (from Section 4) which represents a Gaussian distribution, with **static** parameters assuming natural conjugate prior distributions. (A prior is called conjugate with respect to a likelihood if it takes the same functional form.)

Hence, we can write down the Naive Bayes model (for a simple gender classification task) very succinctly as follows (using the indexed model notation from Section 3).

table People

g	**mod**(2)!**rnd**	**output**	CDiscrete(N=2,R=1.0)
height	**real**!**rnd**	**output**	(CG(M=0.0,P=1.0))[g<2]
weight	**real**!**rnd**	**output**	(CG(M=0.0,P=1.0))[g<2]
footsize	**real**!**rnd**	**output**	(CG(M=0.0,P=1.0))[g<2]
Us	**real**[2][2]!**qry**	**static output**	[[0.0;−20.0];[−10.0;0.0]]
action	**mod**(2)!**qry**	**output**	Action(N=2,UPT=Us,**class**=g)

The first four lines define a Naive Bayes model with Gaussian features height, weight, and footsize, which are assumed to be distributed as independent Gaussians conditional on knowing gender g. At this point, we could simply return the probability vector **infer**.Discrete[2].probs(g): the probabilities that a person has either gender.

However, suppose we need to return a concrete gender decision and that for some reason the cost of false positives differs from the cost of false negatives. Below we encode how to decide whether to take the action of predicting the gender of 0 (female) or 1 (male), given that: A false positive (predict 1 but actually 0) costs 20. A false negative (predict 0 but actually 1) costs 10. A true positive or true negative costs 0. The costs, expressed as negative utilities, are in the matrix Us.

The query defined by the model computes an action column, classifying each row, taking into account the relative costs of false positives and false negatives. (It recommends an action for all rows, even those already labelled with a gender.)

fun Action

N	**int**!**det**	**static input**	
UPT	**real**[N][N]!**qry**	**static input**	
class	**mod**(N)!**rnd**	**input**	
probs	**real**[N]!**qry**	**output**	**infer**.Discrete[N].probs(**class**)
EU	**real**[N]!**qry**	**output**	[**for** p < N →Sum([**for** t < N →(probs[t] * UPT[p][t])])]
ret	**mod**(N)!**qry**	**output**	ArgMax(EU)

We see that the function evaluates N different expected utilities, one for each decision option. ArgMax returns the option delivering the maximum expected utility.

In terms of Bayesian Decision Analysis, the outcome space X is all (predicted class (p), true class (t)) pairs, whose elements are given utilities by UPT. In the expected utility (EU) computations, the Action function only sums over the N outcomes that are consistent with the current p, that is, if the prediction is p, then the probability of any outcome (p',t) where $p' \neq p$ is 0 and can be dropped.

(3) F1 Score: Optimizing a more complex decision criterion. We introduce another model, the Bayes Point Machine, and use it to illustrate a more complicated utility function, namely the F1 score. The F1 score is a measure of accuracy for binary classification that takes into account both false positives and false negatives.

As can be seen from the Tabular code in Figure 1, in table Data (abbreviated here), the data schema consists of seven real-valued clinical measurements X0 to X6 and a Boolean outcome variable Y to be predicted. The model is an instance of the Bayes Point Machine [21], a Bayesian boolean classifier, in which the prior over the weight vector W is drawn from a VectorGaussian, and the label Y is generated by thresholding a noisy score Z which is the inner product between the input vector and the weight vector W. Attribute ProbTrue records the marginal predictive probability for the label, obtained by querying the bias of the Bernoulli random variable Y.

The set D of decision options is given in table Ts, which (we assume) enumerates a number of candidate thresholds Th used to decide the test results by thresholding the marginal predictive probability of each point against Th. For each threshold Th, attribute Decisions is an array, indexed by data point d, containing the candidate decision for d obtained by the thresholding expression d.ProbTrue > Th. The columns ETP, EFP, and EFN evaluate the expected number of true positives, false positives, and false negatives, respectively, by summing the relevant marginal probabilities over test data, which is valid due to linearity of the expectation operator. Finally, the approximate expected F1 score is calculated for each threshold using:

$$E[F_1] = E\left[\frac{2\cdot \mathrm{TP}}{2\cdot \mathrm{TP}+\mathrm{FP}+\mathrm{FN}}\right] \approx \frac{2\cdot E[\mathrm{TP}]}{2\cdot E[\mathrm{TP}]+E[\mathrm{FP}]+E[\mathrm{FN}]}\,.$$

table Data			
X0	real!**det**	**input**	
⋮			
X6	real!**det**	**input**	
Mean	**vector**!**det**	**static output**	VectorFromArray([**for** i < 7 →0.0])
CoVar	PositiveDefiniteMatrix!**det**	**static output**	IdentityScaledBy(7,1.0)
W	**vector**!**rnd**	**static output**	VectorGaussianFromMeanAndVariance(Mean,CoVar)
Z	**real**!**rnd**	**output**	InnerProduct(W,VectorFromArray([X0;X1;X2;X3;X4;X5;X6]))
Y	**bool**!**rnd**	**output**	Gaussian(Z,0.1)> 0.0
ProbTrue	**real**!**qry**	**output**	**infer**.Bernoulli[].Bias(Y)
Train	**bool**!**det**	**input**	
table Ts			
Th	**real**!**det**	**input**	
Decisions	**bool**[SizeOf(Data)]	**output**	[**for** d < SizeOf(Data)→d.ProbTrue > Th]
ETP	**real**!**qry**	**output**	Sum([**for** d < SizeOf(Data)→
			if (!d.Train)& Decisions[d] **then** d.ProbTrue **else** 0.0])
EFP	**real**!**qry**	**output**	Sum([**for** d < SizeOf(Data)→
			if (!d.Train)& Decisions[d] **then** 1.0 − d.ProbTrue **else** 0.0])
EFN	**real**!**qry**	**output**	Sum([**for** d < SizeOf(Data)→
			if (!d.Train)& (!Decisions[d])**then** d.ProbTrue **else** 0.0])
EF1	**real**!**qry**	**output**	(2.0 ∗ ETP)/ ((2.0 ∗ ETP)+ EFP + EFN)
table Decisions			
ChosenThID	**link**(Ts)!**qry**	**static output**	ArgMax([**for** t < SizeOf(Ts)→t.EF1])
ChosenTh	**real**!**qry**	**static output**	ChosenThID.Th
DataID	**link**(Data)!**det**	**input**	
Decision	**bool**!**qry**	**output**	ChosenThID.Decisions[DataID]

Fig. 1. F1 computation in Tabular on mammography data

This is an approximation because the F1 score is a non-linear function in TP, FP, and FN, and is employed here because it allows us to express the expectation in terms of marginal probabilities which are available from our inference back end. Recent work by Nowozin [24] has shown that approximations of this form yield good results.

The final table Decisions determines the optimal threshold ChosenTh by finding the identity of the threshold t that maximises t.EF1. In addition, Decisions outputs, for each data point DataID, the labelling Decision obtained with the optimal threshold (assuming that column DataID enumerates the keys of table Data).

7 Tabular Excel: Implementing Tabular in a Spreadsheet

Public releases of the Tabular add-in for Excel are available from `http://research.microsoft.com/tabular`. The add-in extends Excel with a new task pane for authoring models, running inference and setting parameters of Infer.NET. A user authors the model within a rectangular area of a worksheet. Tabular parses and type-checks the model in the background, enabling the inference button when the model is well-typed. Tabular pulls the data schema and data itself from the relational Data Model of Excel 2013. The results of inference and queries are then reported back to the user as augmented Excel tables. Tabular Excel is able to concisely express a wide range of models beyond those illustrated here (see companion technical report [15]).

Type checking the Tabular schema results in a type-annotated schema. This is elaborated to core form, eliminating all function calls and indexed models. The core schema is then translated to an Infer.NET [20] factor graph, constructed dynamically with

Infer.NET's (imperative and weakly typed) modelling API. Our (type-directed) translation relies on and exploits the fact that all table sizes are known and that discrete random variables, which may be used to index into arrays, have known support. Moreover, the space of any (explicit or implicit) array indexing expression is used to insert the requisite Infer.NET *switch* construct when indexing through a **rnd**-space index (as demonstrated in an appendix to technical report [15]). The fruits of Infer.NET inference are approximate marginal distributions for the **rnd**-space bindings of the schema. Expressions in **det** and **qry**-space are evaluated by interpretation after inference, binding input to the concrete data and **rnd**-level variables to their inferred distributions. Thus **qry**-space expressions have access to the inputs, deterministic values and distributions on which they depend. For compilation, the type system ensures that the value of **qry**-space expression cannot depend on the particular value of a **rnd**-space variable (only its distribution) and that all **rnd**-space variables can be inferred prior to **qry** evaluation.

Users can also extract C# source code to compile and run their models outside Excel (see [15] for an example). This supports subsequent customization by Infer.NET experts as well as integration in standalone applications. One of our internal users has extracted code in this way to perform inference on a large dataset with approximately 42 million entities and 46 million relationships between them. Inference required 7.5 hours of processing time on a 2-core Intel Xenon L5640 server with 96 GB of RAM. The extracted code is also useful for debugging compilation and applications that need to separate learning (on training data) from prediction (on new data).

The following is direct comparison between the Tabular Excel form of the Mammography model (Figure 1) with code for the same problem written in C# using Infer.NET. We get the same statistical answers in both cases, though there are differences in code speed. Initially, Tabular queries were (naively) interpreted, not compiled; adopting simple runtime code generation techniques has allowed us to reduce the **qry** time from 1601ms (interpreted) to 29ms (compiled), a 55x fold increase. The handwritten C# model is slower on inference because it is effectively compiled and run twice, once for training and another time for prediction.

	data (LOC)	model (LOC)	decisions (LOC)	inference (ms)	query (ms)
Infer.NET	35	35	45	2968	6
Tabular	0	15	14	1529	1601/29

8 Related Work

Interest in probabilistic programming languages is rising as evinced by recent languages like Church [10], a Turing-complete probabilistic Scheme with inference based on sampling, and its relatives Anglican [29] (a typed re-imagination of Church) and Venture [18] (a variant of Church offering programmable inference). Other recent works include R2 [23], which uses program analysis to optimize MCMC sampling of probabilistic programming, Uncertain<T> [3], a simple abstraction for embedding probabilistic reasoning into conventional programs that handle uncertain data, and Wolfe [27], where inference is expressed within a host language by providing a small set of primitives for writing distributons and operations for maximization and summation.

To the best of our knowledge, few systems offer explicit support for decision theory. IBAL's [25] impressive framework aims to combine Bayesian inference and decision theory "under a single coherent semantic framework". IBAL makes use of query information and only computes the quantities needed to answer specific queries. Other systems that extend probabilistic languages with dedicated decision theoretic constructs are described in [6, 4, 22]. The main difference in our approach is that while our post-processing can be used to implement decision theory strategies, decision theoretic constructs are not built into the language. This is a pragmatic choice. In general, decision theory involves two intractabilities: computing expected utilities, and optimizing over the decisions. IBAL and DT-ProbLog [4] have some general-purpose approximations, but often problem-specific approximations are needed as in our F1 optimization example or in [24]. It is not clear how these approximations fit into the above frameworks. Tabular's free-form post-processing design allows such bespoke approximations.

STAN [28] allows for post-processing of inference results, but only via separately declared code blocks, rather than being conveniently mingled with the model or abstracted in functions. Although STAN's facilities are expressive and can include arbitrary deterministic and stochastic computations, they are restricted to computing *per sample* quantities. In Tabular terms, this would correspond to computations restricted to **rnd**-space which prevents the computation of the aggregate **qry**-space quantities required for Bayesian decision theory.

Figaro [26] supports post-inference decision-making, but via separate, decision-specific language features, outside the core modelling language. Tabular, instead, uses types to distinguish between operations available in different spaces (or phases) (such as random draws in **rnd** space, optimization (ArgMax) and moments of distributions in **qry**-space). Embedded DSLs such as Infer.NET [20], HANSEI [17] and FACTORIE [19] enable arbitrary post-processing in the host language, but require knowledge of both the host and the embedded language, which is typically much simpler.

Tabular is, to our knowledge, the first probabilistic programming language with dependently typed abstractions. STAN and BUGS [9] do have value-indexed types, but cannot abstract over indexes appearing in types.

We advocate types to help catch errors in probabilistic queries on spreadsheets. There is a body of work on testing and discovering errors on spreadsheets. For example, Ahmad et al. [1] propose unit-based types as a means of catching errors. To the best of our knowledge, dependent types have not previously been applied to spreadsheets.

9 Conclusions

We recast Tabular as a query language on databases held in spreadsheets.

This paper presents a technical evaluation of the design consisting of theorems about its metatheory, demonstration of its expressiveness by example, and some numeric comparisons with the alternative of writing models directly in Infer.NET. Evaluating the usability by spreadsheet users is important, but we leave that task for future work.

We have in mind several lines of future development. One limitation of our current system is that data is modelled by map-style loops over data; to model time-series, it would be useful to add some form of iterative fold-style loops. Another limitation is that programs involve a single run of the underlying inference system: **rnd** space determines the model and its conditioning, and **qry** space determines how the results are processed. To support multiple runs of inference we might consider an indexed hierarchy of spaces where **infer** moves data from $\mathbf{rnd}_i$ space to $\mathbf{qry}_i$ space, and $\mathbf{rnd}_{i+1}$ space can depend on results computed in $\mathbf{qry}_i$ space.

Finally, our approach could be applied to add user-defined functions to languages such as BUGS or Stan, or to design typed forms of universal probabilistic languages such as those in the Church family.

Acknowledgement. Dylan Hutchison commented on a draft. We thank Natalia Larios Delgado and Matthew Smith for their feedback on our Excel addin.

References

[1] Ahmad, Y., Antoniu, T., Goldwater, S., Krishnamurthi, S.: A type system for statically detecting spreadsheet errors. In: 18th IEEE International Conference on Automated Software Engineering (ASE 2003), pp. 174–183 (2003)

[2] Borgström, J., Gordon, A.D., Greenberg, M., Margetson, J., Gael, J.V.: Measure transformer semantics for Bayesian machine learning. Logical Methods in Computer Science 9(3) (2013) preliminary version at ESOP 2011

[3] Bornholt, J., Mytkowicz, T., McKinley, K.S.: Uncertain<T>: A first-order type for uncertain data. In: Architectural Support for Programming Languages and Operating Systems (ASPLOS) (March 2014)

[4] Van den Broeck, G., Thon, I., van Otterlo, M., De Raedt, L.: DTProbLog: A decision-theoretic probabilistic Prolog. In: AAAI (2010)

[5] Cardelli, L.: Typeful programming. Tech. Rep. 52. Digital SRC (1989)

[6] Chen, J., Muggleton, S.: Decision-theoretic logic programs. In: Proceedings of ILP, p. 136 (2009)

[7] Duda, R.O., Hart, P.E.: Pattern Classification and Scene Analysis. John Wiley & Sons, New York (1973)

[8] Gelman, A., Carlin, J.B., Stern, H.S., Dunson, D.B., Vehtari, A., Rubin, D.B.: Bayesian Data Analysis, 3rd edn. Chapman & Hall (2014)

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

[10] Goodman, N., Mansinghka, V.K., Roy, D.M., Bonawitz, K., Tenenbaum, J.B.: Church: a language for generative models. In: Uncertainty in Artificial Intelligence (UAI 2008), pp. 220–229. AUAI Press (2008)

[11] Goodman, N.D.: The principles and practice of probabilistic programming. In: Principles of Programming Languages (POPL 2013), pp. 399–402 (2013)

[12] Gordon, A.D., Aizatulin, M., Borgström, J., Claret, G., Graepel, T., Nori, A., Rajamani, S., Russo, C.: A model-learner pattern for Bayesian reasoning. In: POPL (2013)

[13] Gordon, A.D., Graepel, T., Rolland, N., Russo, C.V., Borgström, J., Guiver, J.: Tabular: a schema-driven probabilistic programming language. In: POPL (2014a)

[14] Gordon, A.D., Henzinger, T.A., Nori, A.V., Rajamani, S.K.: Probabilistic programming. In: Future of Software Engineering (FOSE 2014), pp. 167–181 (2014b)

[15] Gordon, A.D., Russo, C., Szymczak, M., Borgström, J., Rolland, N., Graepel, T., Tarlow, D.: Probabilistic programs as spreadsheet queries. Tech. Rep. MSR–TR–2014–135, Microsoft Research (2014c)

[16] Herbrich, R., Minka, T., Graepel, T.: TrueSkill^{tm}: A Bayesian skill rating system. In: Advances in Neural Information Processing Systems, NIPS 2006 (2006)

[17] Kiselyov, O., Shan, C.: Embedded probabilistic programming. In: Conference on Domain-Specific Languages, pp. 360–384 (2009)

[18] Mansinghka, V., Selsam, D., Perov, Y.: Venture: a higher-order probabilistic programming platform with programmable inference. arXiv preprint arXiv:1404.0099 (2014)

[19] McCallum, A., Schultz, K., Singh, S.: Factorie: Probabilistic programming via imperatively defined factor graphs. In: NIPS 2009, pp. 1249–1257 (2009)

[20] Minka, T., Winn, J., Guiver, J., Knowles, D.: Infer.NET 2.5 (2012), Microsoft Research Cambridge. http://research.microsoft.com/infernet

[21] Minka, T.P.: A family of algorithms for approximate Bayesian inference. Ph.D. thesis, Massachusetts Institute of Technology (2001)

[22] Nath, A., Domingos, P.: A language for relational decision theory. In: Proceedings of the International Workshop on Statistical Relational Learning (2009)

[23] Nori, A.V., Hur, C.K., Rajamani, S.K., Samuel, S.: R2: An efficient MCMC sampler for probabilistic programs. In: Conference on Artificial Intelligence, AAAI (July 2014)

[24] Nowozin, S.: Optimal decisions from probabilistic models: the intersection-over-union case. In: Proceedings of CVPR 2014 (2014)

[25] Pfeffer, A.: The design and implementation of IBAL: A general-purpose probabilistic language. In: Getoor, L., Taskar, B. (eds.) Introduction to Statistical Relational Learning. MIT Press (2007)

[26] Pfeffer, A.: Figaro: An object-oriented probabilistic programming language. Tech. rep., Charles River Analytics (2009)

[27] Riedel, S.R., Singh, S., Srikumar, V., Rocktäschel, T., Visengeriyeva, L., Noessner, J.: WOLFE: strength reduction and approximate programming for probabilistic programming. In: Statistical Relational Artificial Intelligence (2014)

[28] Stan Development Team: Stan: A C++ library for probability and sampling, version 2.2 (2014), `http://mc-stan.org/`

[29] Wood, F., van de Meent, J.W., Mansinghka, V.: A new approach to probabilistic programming inference. In: Proceedings of the 17th International conference on Artificial Intelligence and Statistics (2014)

[30] Xi, H., Pfenning, F.: Eliminating array bound checking through dependent types. In: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI), pp. 249–257 (1998)