14

Tabular: Probabilistic Inference from the Spreadsheet

Andrew D. Gordon
Microsoft Research and University of Edinburgh
Claudio Russo[a]
DFINITY, Zürich
Marcin Szymczak[b]
RWTH Aachen University
Johannes Borgström
Uppsala University
Nicolas Rolland[a] and Thore Graepel[a]
University College London
Daniel Tarlow[a]
Google Research, Brain Team, Montreal

Abstract: Tabular is a domain-specific language for expressing probabilistic models of relational data. Tabular has several features that set it apart from other probabilistic programming languages including: (1) programs and data are stored as spreadsheet tables; (2) programs consist of probabilistic annotations on the relational schema of the data; and (3) inference returns estimations of missing values and latent columns, as well as parameters. Our primary implementation is for Microsoft Excel and relies on Infer.NET for inference. Still, the language can be called independently of Excel and can target alternative inference engines.

14.1 Overview

Probabilistic programming languages promise to make machine learning more accessible by allowing users to write their generative models as computer programs and providing generic inference engines capable of performing inference on all valid programs expressible in the given language. However, as most of the currently existing languages are essentially probabilistic extensions of conventional programming languages, they are arguably not ideally suited for the job.

For one thing, they are still difficult to use for people who are not professional programmers. Meanwhile, many people who may want to use probabilistic mod-

[a] The work was conducted while the authors were at Microsoft Research

[b] The work was conducted while the author was at University of Edinburgh and Oxford University

[c] From *Foundations of Probabilistic Programming*, edited by Gilles Barthe, Joost-Pieter Katoen and Alexandra Silva published 2020 by Cambridge University Press.

https://doi.org/10.1017/9781108770750.015 Published online by Cambridge University Press

elling are domain experts, for instance business analysts, who often have limited programming experience and could find, for instance, systems based on functional (Goodman et al., 2008; Wood et al., 2014) or logic (Van den Broeck et al., 2010) programming baffling. Secondly, most existing languages require all the necessary data to be loaded and placed in the right data structures. This can often be problematic and require a large amount of data pre-processing, which could be a nuisance even for experienced programmers and statisticians.

The Tabular language, first presented by Gordon et al. (2014b), takes a different approach. Instead of extending an ordinary programming language with primitives for sampling and conditioning, Tabular extends schemas of relational databases with probabilistic model expressions and annotations. This idea is based on the observation that in model-based Bayesian machine learning, the starting point is not the model itself, but the dataset to which one wants to fit a model, which has to be stored in some sort of database – for example a spreadsheet. In Tabular, the probabilistic model is built on top of the data, and the input database does not need to be manipulated before being fed to the program.

A key strength of Tabular is that it is easier to use than standard programming languages, because it does not require the user to write an actual program from scratch – all that the user has to do to define a model is to annotate a database schema with probabilistic expressions explaining how they believe the data was generated and add latent columns for the unknown quantities of interest. Moreover, Tabular's design allows it to be integrated with environments such as spreadsheet applications, that are familiar to users who are not professional programmers. Indeed, Tabular has been implemented as an Excel plugin and both the model and the input database are specified as Excel spreadsheets. Inference results are also saved to a spreadsheet, which allows for easier post-processing and visualisation. The command-line version of Tabular is open source and the source code is available at `https://github.com/TabularLang/CoreTabular`.

In this chapter, we present a new, substantially enhanced version of Tabular, which features user-defined functions and queries on inference results. We endow Tabular with a structural, dependent type system, which helps understand the sample space of a program and catch common modelling mistakes. We define a reduction relation reducing Tabular programs with function applications to Core models containing only simple expressions and corresponding directly to factor graphs. We also demonstrate by example how these features make Tabular a useful language for Bayesian modelling.

This chapter is based on Chapter 4 of Szymczak (2018), which is itself an extended version of Gordon et al. (2015).

https://doi.org/10.1017/9781108770750.015 Published online by Cambridge University Press

14.2 Introduction and Examples

In this section, we introduce Tabular informally, explaining its features by example.

14.2.1 Probabilistic Programming in Tabular

A Tabular program is constructed by extending a database schema with:

- *Latent columns* representing unknown parameters, not present in the database, which we want to infer from the data,
- *Annotations* defining roles of respective columns in the probabilistic model (input variables, modelled output variables, local variables),
- *Model expressions*, which express our belief about how the values in the given column of the database were generated.

In the simplest case, model expressions are ordinary expressions written in a first-order functional language with random draws. We refer to schemas and tables containing only such simple expressions as Core schemas and tables. Other kinds of models include function applications and indexed models, which will be discussed later.

Let us begin the presentation of Tabular with an example based on one from Gordon et al. (2014b), which implements the TrueSkill model (Herbrich et al., 2007) for ranking players in online video games. Suppose we have a database containing the outcomes of past matches between some players. This database can have the following schema (where we assume that each table has an implicit, integer-valued ID column, serving as the primary key of the table):

table Players	
Name	**string**
table Matches	
Player1	**link**(Players)
Player2	**link**(Players)
Win1	**bool**

where Win1 is true if the match was won by Player 1 and false if Player 2 won the match (we assume there are no draws). Based on these past results, we want to infer the relative skills of the players.

According to the TrueSkill model, we quantify the *performance* of a given player in a certain match by a numeric value, which is a noisy copy of the player's skill. We assume that each match was won by the player with higher performance value. We can implement this model in Tabular by extending the above schema as follows[1]:

[1] As explained in Section 14.3, in the formal syntax of Tabular, each column has a global and local name, because of issues with α-conversion. In the introductory examples in this section, we only give each column one name, serving both as a global and local identifier, to simplify presentation.

https://doi.org/10.1017/9781108770750.015 Published online by Cambridge University Press

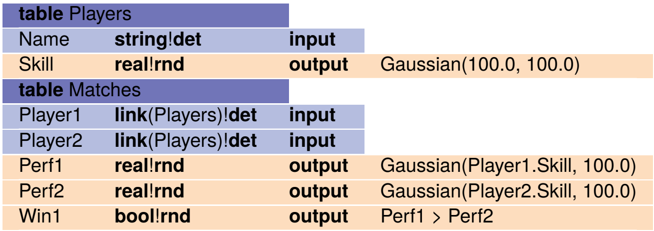


table Players			
Name	**string**!**det**	**input**	
Skill	**real**!**rnd**	**output**	Gaussian(100.0, 100.0)
table Matches			
Player1	**link**(Players)!**det**	**input**	
Player2	**link**(Players)!**det**	**input**	
Perf1	**real**!**rnd**	**output**	Gaussian(Player1.Skill, 100.0)
Perf2	**real**!**rnd**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

We have added one new column, not present in the database, to the Players table and two columns to the Matches table. The Players table now has a Skill attribute. This column is not expected to be present in the input database – its distribution is to be inferred from the observed data. By assigning the expression Gaussian(100.0, 100.0) to this column, we have defined the prior distribution on players' skills to be a Gaussian with mean 100 and variance 100. Similarly, the values of the Perf1 and Perf2 columns are, in the generative interpretation of the model, drawn from Gaussians centred at the skills of the corresponding players (the expression Player1.Skill is a reference to the value of Skill in the row of Players linked to by Player1, and similarly for Player2.Skill). Finally, the observed Win1 column is assigned the expression Perf1 > Perf2, which expresses the condition that in every row of the Matches table, Perf1 must be greater than Perf2 if Win1 in this row is **true** in the database, and not greater than Perf2 if Win1 is **false** – otherwise, the values of the parameters would be inconsistent with the observations.

The types in the second schema include **det** and **rnd** annotations which specify whether the data in the given column is deterministic (known in advance) or random (to be inferred by the inference algorithm). These annotations, which we call *spaces*, are used by the type system to catch information flow errors, such as supposedly deterministic data depending on random variables. Tabular columns can also be in space **qry**, which will be discussed later.

To perform inference in the above model, we need to parametrise it on a particular dataset. In Tabular, like in BUGS (Gilks et al., 1994) and Stan (Carpenter et al., 2017), input data is decoupled from the program and is loaded by the compiler from a separate data source. This approach makes it possible to run inference in the same model with multiple datasets without modifying the model. The TrueSkill model, as implemented above, was designed to be applied to databases containing thousands of matches and players, but the following is a valid tiny input database for this schema:

Players

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

Matches

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

https://doi.org/10.1017/9781108770750.015 Published online by Cambridge University Press

In this example, we have only three players, Alice, Bob and Cynthia, and we assume that Bob beat Alice in the first match and was beaten by Cynthia in the second one.

The default inference algorithm of Tabular, Expectation Propagation (Minka, 2001), adds the approximate distributions of unobserved random columns to the input database. The output database for the above tiny example is as follows:

Players

ID	Name	Skill
0	`"Alice"`	Gaussian(95.25, 82.28)
1	`"Bob"`	Gaussian(100.0, 70.66)
2	`"Cynthia"`	Gaussian(104.8, 82.28)

Matches

ID	P1	P2	Perf1	Perf2	Win1
0	0	1	Gaussian(90.49, 129.1)	Gaussian(104.8, 123.6)	**false**
1	1	2	Gaussian(95.25, 123.6)	Gaussian(109.5, 129.1)	**false**

This matches our intuition that Cynthia, having beaten the winner of the first match, is most likely to be the best of the three players, and Alice is probably the weakest.

In addition to the style of inference described above, called *query-by-latent-column*, Tabular also supports *query-by-missing-value*, where the database has some missing entries for one or many **output** columns and the goal is to compute the distributions on the missing values. For example, if we want to predict the outcome of an upcoming match between Alice and Cynthia, we can extend the matches table as follows:

Matches

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

The Tabular inference engine will then compute the distribution of Win1 in the third column.

Matches

ID	P1	P2	Perf1	Perf2	Win1
0	0	1	Gaussian(90.49, 129.1)	Gaussian(104.8, 123.6)	**false**
1	1	2	Gaussian(95.25, 123.6)	Gaussian(109.5, 129.1)	**false**
2	0	2	Gaussian(95.25, 182.3)	Gaussian(104.8, 182.3)	Bernoulli(0.3092)

https://doi.org/10.1017/9781108770750.015 Published online by Cambridge University Press

14.2.2 User-Defined, Dependently-Typed Functions

Tabular supports *functions*, which are defined in the same way as ordinary tables and can be used to abstract away arbitrary repeated blocks of code which only differ by some values used in the model expressions. Functions can help users make their schemas shorter and more concise. Tabular already comes with a library of predefined functions, representing, for instance, commonly used conjugate models, and new functions can be defined by the user.

To illustrate how functions can be used in Tabular, let us consider the well-known problem of inferring the bias of a coin from the outcomes of coin tosses. Assuming that each bias (between 0 and 1) is equally likely, this model can be represented in Tabular as follows:

table Coins			
V	**real**!**rnd**[2]	**static output**	Dirichlet[2]([1.0, 1.0])
Flip	**mod**(2)!**rnd**	**output**	Discrete[2](V)

where Dirichlet[2]([1.0, 1.0]) is just the uniform distribution on pairs of two probabilities adding up to 1, and Discrete[2](V) draws 0 or 1 (representing tails and heads, respectively) with probability proportional to the corresponding component of V.

This model, in which the parameter to the discrete distribution has a uniform Dirichlet prior, is an instance of the *Conjugate Discrete* model. Conjugate Discrete, which is a building block of many more complex models, is defined in the standard function library as follows:

fun CDiscrete			
N	**int**!**det**	**static input**	
R	**real**!**det**	**static input**	
V	**real**!**rnd**[N]	**static output**	Dirichlet[N]([**for** i < N → R])
ret	**mod**(N)!**rnd**	**output**	Discrete[N](V)

The arguments of this function, N and R, denote, respectively, the length of the parameter vector and the value of each component of the hyperparameter vector passed to the prior (the higher the value of R, the closer together the components of the parameter vector are expected to be). This function also demonstrates the use of dependent types: **real**!**rnd**[N] indicates that the given random column is an array of reals of size determined by the variable N, and **mod**(N)!**rnd** denotes a non-negative random integer smaller than N. It is worth noting that in the definition of CDiscrete we could alternatively make the entire pseudocount vector passed to Dirichlet[N] an argument of type **real**!**det**[N].

With this function in place, we can rewrite the coin toss model as follows:

table Coins			
Flip	**mod**(2)!**rnd**	**output**	CDiscrete(N=2, R=1.0)

where setting R to 1.0 guarantees that the prior distribution on probabilities V of heads and tails is uniform.

https://doi.org/10.1017/9781108770750.015 Published online by Cambridge University Press

The reduction algorithm presented later in this chapter reduces this table to the form shown earlier, modulo renaming of column names.

Tabular also supports *indexing* function applications, which results in turning static parameters of the model into arrays, indexed by a categorical variable (that is, a discrete random variable with a finite domain). For example, suppose that in the above problem we have two coins with different biases, and we always toss one of them, chosen at random with equal probability. To infer the biases of the coins, we can adapt the above Tabular program as follows:

table Coins			
CoinUsed	**mod**(2)!**rnd**	**output**	Discrete[2]([0.5, 0.5])
Flip	**int**!**rnd**	**output**	CDiscrete(N=2, R=1.0)[CoinUsed < 2]

Now, we have two copies of the bias vector V, one for each coin, and at each row, the vector indicated by the random variable CoinUsed is used.

14.2.3 *Query Variables*

Another novel feature of Tabular is the **infer** operator, which can be used to extract properties of an inferred distribution, such as its mean (in case of, say, a Gaussian) or bias (in case of a Bernoulli distribution). These properties can then be used to compute some pseudo-deterministic data dependent on the inference results.

For instance, in the above biased coin example, we might be interested in extracting the actual bias of the coin, as a numeric value rather than a distribution. Since the posterior distribution of the bias is a Dirichlet distribution, parametrized by the "pseudocounts" of the numbers of heads and tails, the bias itself is the count of heads divided by the sum of the counts. Using the **infer** operator, we can compute it as follows:

table Coins			
V	**real**!**rnd**[2]	**static output**	Dirichlet[2]([1.0, 1.0])
Flip	**mod**(2)!**rnd**	**output**	Discrete[2](V)
counts	**real**!**qry**[2]	**static local**	**infer**.Dirichlet[2].pseudocount(V)
Bias	**real**!**qry**	**static output**	counts[1]/(counts[1]+counts[0])

For instance, if we apply this model to a tiny database consisting of three coin flip outcomes, two of them being heads and one being tails, the inference algorithm returns the following static quantities:

Coins

V	counts	Bias
Dirichlet(2, 3)	[2,3]	0.6

In the expression **infer**.Dirichlet[2].pseudocount(V), Dirichlet[2] denotes the type of distribution from which we want to extract a property, pseudocount is the name of the parameter we want to extract (in Tabular, all distributions have named parameters) and V is the column in which the distribution is defined.

All columns containing calculations dependent on the result of a query are in the **qry** space. Columns in this space can only reference random variables via the **infer** operator.

After adding the **infer** operator, we now have three different kinds of columns in Tabular: deterministic columns, whose values are known before inference; random columns, whose distributions are to be inferred and may depend on deterministic columns, and query columns, depending on inferred distributions. The values or distributions of these columns (in all rows) must be computed in the right order, for instance, a random column cannot depend on the result of a query. To make sure that there are no erroneous dependencies in the program, the columns are split into three spaces: **det**, **rnd** and **qry**; space annotations ensure that the constraints on dependencies between columns are preserved.

14.2.4 Related Work

Probabilistic programming is becoming an increasingly popular approach to Bayesian inference and many new languages following different paradigms were created recently. These include functional languages like Fun (Borgström et al., 2013), Church (Goodman et al., 2008), Anglican (Wood et al., 2014), Venture (Mansinghka et al., 2014), WebPPL (Goodman and Stuhlmüller, 2014) and monad-bayes (Ścibior et al., 2015), procedural languages like R2 (Nori et al., 2014), Infer.NET (Minka et al., 2012) and Stan (Carpenter et al., 2017), logical languages such as ProbLog (Van den Broeck et al., 2010) and even an implementation of the probabilistic process algebra ProPPA (Georgoulas et al., 2014). Recently, a new class of probabilistic languages marrying Bayesian modelling with deep neural networks saw the light of day. These include Pyro (Bingham et al., 2018) and ProbTorch (Siddharth et al., 2017).

Designing a language involves a trade-off between expressiveness and performance. In this respect, probabilistic languages can be roughly divided into two groups: universal, Turing-complete languages such as R2 and Church and its descendants, which allow creation of arbitrary probabilistic models (including non-parametric models with unbounded numbers of random variables) but can only use a limited range of sampling-based inference algorithms, and more restricted languages like Infer.NET and BUGS (Gilks et al., 1994), in which models correspond to factor graphs and which can therefore use a wider class of inference algorithms, including algorithms for factor graphs. Tabular belongs to the second class and uses Expectation Propagation (Minka, 2001) as its default inference algorithm.

In terms of the paradigm and user interface, the two probabilistic programming

https://doi.org/10.1017/9781108770750.015 Published online by Cambridge University Press

packages most related to Tabular are BayesDB (Mansinghka et al., 2015) and Scenarios (Wu et al., 2016). BayesDB is a probabilistic package based on relational database schemas. Its modelling language is a probabilistic extension of SQL, which is significantly different from the approach taken by Tabular, where models are written by directly annotating database schemas. Scenarios is a commercial Excel plugin which allows defining probabilistic models in a spreadsheet environment. The difference between Scenarios and Tabular is that the former is shallowly embedded in Excel and defines probabilistic models via special Excel functions, while Tabular is a standalone probabilistic language which simply uses Excel as a convenient development environment. Because of its shallow embedding within a dynamically typed formula language, Scenarios does not have a static type system.

14.2.5 Retrospective and Related Projects

Tabular was first presented by Gordon et al. (2014b) as a language based on database schemas, with a standalone implementation in the form of a GUI program interfacing directly with a relational database. This initial version of the language did not support functions – models were either simple expressions or predefined conjugate models from a small, fixed library. Tabular had a non-dependent type system without spaces, in which the type of a schema was a quintuple of nested record types, whose components specified the types of hyperparameters, parameters, inputs and latent and observed variables of the model defined by the table. The semantics of Tabular was defined by means of translation to Fun (Borgström et al., 2013), a first-order, functional probabilistic programming language. Post-processing of inference results had to be done outside Tabular.

A revised version of the language, described by Gordon et al. (2015), added support for user-defined functions and queries, as described earlier in this section. The original type system was replaced by a simpler one, in which types themselves have a similar form to Core Tabular tables. The new type system supports basic dependent types and provides space annotations dividing columns into deterministic, random and query columns. The semantics of the new version of Tabular consists of a reduction system reducing schemas with functions and indexing to the Core form and a semantics of Core Tabular models (omitted in this chapter, see the long version of the aforementioned paper (Gordon et al., 2014a)), defined directly in terms of measure theory. Reduction to Core form is proven type-sound.

Further improvements to Tabular were presented in the doctoral dissertation of Szymczak (2018), which introduced double column names to fix a problem with α-conversion and presented a more rigorous proof of type soundness of schema reduction. The dissertation also described a new, arguably more rigorous and elegant, semantics of Core Tabular models.

https://doi.org/10.1017/9781108770750.015 Published online by Cambridge University Press

Fabular, presented by Borgström et al. (2016), extends Tabular with hierarchical linear regression formulas, extending the formula notation used by R packages such as `lmer`. Such formulas allow for a concise representation of a wide class of models and can be used in Tabular like any other model expressions.

Additionally in his Master's dissertation, Hutchison (2016) presented a generative grammar allowing dynamic creation of Tabular programs, which could serve as a basis for an automated model suggestion tool for Tabular. Hence, Tabular was used as part of a study of the internet-based trade in specimens of endangered species of plants (Vaglica et al., 2017).

14.3 Syntax of Tabular

Having introduced Tabular informally, we now present the formal syntax of the language. Since programs and data are decoupled in Tabular, we need to define the syntax for both Tabular databases and schemas.

14.3.1 Syntax of Databases

A Tabular database is a tuple $DB = (\delta_{in}, \rho_{sz})$, consisting of two maps whose domain is the set of names of tables in the database. The first map, $\delta_{in} = [t_i \mapsto \tau_i{}^{i \in 1..n}]$, assigns to each table another map $\tau_i = [c_j \mapsto a_j{}^{j \in 1..m_i}]$ mapping each column c_i to an attribute a_i. An *attribute* $a_j = \ell_j(V_j)$ consists of a *level* ℓ_j and a *value* V_j, which can be a scalar s (that is, an integer, a real or a Boolean) or an array of values. The level of an attribute can be either **static**, in which case the given column has only one value accross all rows, or **inst**, which means that the column has one value per row. In the latter case, V_j is actually an array of values, with one value per row. Column names c_j have the same form as external column names in schemas (described below), except that they are not allowed to be empty.

The second map, $\rho_{sz} = [t_i \mapsto sz_i{}^{\ i \in 1..n}]$, simply stores the sizes of tables. The value of each **inst**-level attribute of table t_i must be an array of size sz_i.

Any value V_j in the database can be *nullable*, that is, any **static** attribute can have an empty value (denoted ?) and in any **inst** attribute, any number of component values can be empty. An empty value in a row of an **output** column means that the distribution on the given row and column is to be inferred from other data by the inference algorithm.

Databases, Tables, Attributes, and Values:

$\delta_{in} ::= [t_i \mapsto \tau_i{}^{\ i \in 1..n}]$	table map
$c, o ::= b_1.(\ldots).b_n$	column name
$\rho_{sz} ::= [t_i \mapsto sz_i{}^{\ i \in 1..n}]$	table size map

https://doi.org/10.1017/9781108770750.015 Published online by Cambridge University Press

$a ::= \ell(V)$	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 ($\textbf{static} < \textbf{inst}$)
$\tau ::= [c_j \mapsto a_j \;{}^{j \in 1..m}]$	table in database

For example, the input database for the TrueSkill example from Section 14.2.1 can be written as follows using the formal syntax of databases:

$$
\begin{aligned}
&[\mathsf{Players} \mapsto [\mathsf{ID} \mapsto \textbf{inst}([0,1,2])],\\
&\mathsf{Matches} \mapsto [\mathsf{ID} \mapsto \textbf{inst}([0,1,2]), \mathsf{Player1} \mapsto \textbf{inst}([0,1,0]),\\
&\qquad\qquad \mathsf{Player2} \mapsto \textbf{inst}([1,2,2]), \mathsf{Win1} \mapsto \textbf{inst}([0,0,?])]
\end{aligned}
$$

where player names are omitted (they are insignificant for the model, and the formal syntax of Tabular does not allow strings[2]) and **true** and **false** are represented by 1 and 0, respectively.

14.3.2 Syntax of Core Schemas

We begin by giving the syntax of Core schemas, which have a straightforward interpretation as factor graphs. We first define the basic building blocks of a Tabular column.

Index Expressions, Spaces and Dependent Types of Tabular:

$e ::=$	index expression
$\quad x$	variable
$\quad s$	scalar constant
$\quad \textbf{sizeof}(t)$	size of a table
$S ::= \textbf{bool} \mid \textbf{int} \mid \textbf{real}$	scalar type
$spc ::= \textbf{det} \mid \textbf{rnd} \mid \textbf{qry}$	space
$T, U ::= (S \,!\, spc) \mid (\textbf{mod}(e) \,!\, spc) \mid T[e]$	(attribute) type
$c, o ::= \_ \mid b_1.(\ldots).b_n$	external column name

$$\mathrm{space}(S \,!\, spc) \triangleq spc \qquad \mathrm{space}(\textbf{mod}(e) \,!\, spc) \triangleq spc \qquad \mathrm{space}(T[e]) \triangleq \mathrm{space}(T)$$

An *index expression* is a constant, a variable (referencing a previous column or an array index) or a **sizeof** expression, returning the size of the given table (that is, $\textbf{sizeof}(t)$ returns $\rho_{sz}(t)$ if ρ_{sz} is the map of table sizes). A *scalar type* is one of **bool**, **int** or **real**. These correspond to scalar types in conventional languages. A *space* of a column, being part of its type, can be either **det**, **rnd** or **qry**, depending on

[2] The implementation of Tabular does support strings and implicitly converts them to integers

whether the column is deterministic, random or at query-level. An attribute *type* can be either a scalar type S with a space, a dependent bounded integer type **mod**(e), whose bound is defined by the indexed expression e, with a space, or a recursively defined array type $T[e]$, where T is an arbitrary type and e an index expression defining the size of the array. We use **link**(t) as a shorthand for **mod**(**sizeof**(t)). An *external column name*, used to reference a column from another table or to access a field of a reduced function body, is either empty (denoted by _) or consists of a sequence of one or more *atomic names* b_i, separated by dots.

The space operator, used in the remainder of this chapter, returns the unique space annotation nested within the given type.

As an example, consider the type of the Flip column in the Coins table in Section 14.2.2, **mod**(2)!**rnd**. This is a type of random non-negative integer-valued expressions bounded by 2 (that is, admitting only values 0 and 1). It is clear by definition of space that space(**mod**(2)!**rnd**) = **rnd**. Moreover, the type of the V column in CDiscrete in Section 14.2.2, **real**!**rnd**[N], is a type of arrays of size N (where the variable N represents a deterministic non-negative integer), whose elements all have type **real**!**rnd** (that is, are random real-valued expressions). The space of this type is given by space(**real**!**rnd**[N]) = space(**real**!**rnd**) = **rnd**.

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
if E **then** F_1 **else** F_2	if-then-else
$[E_1, \ldots, E_n] \mid E[F]$	array literal, lookup
[**for** $x < e \rightarrow F$]	for loop (scope of index x is F)
infer.$D[e_1, \ldots, e_m].c(E)$	parameter c of inferred marginal of E
$E : t.c$	dereference link E to instance of c
$t.c$	dereference static attribute c of t

This grammar of expressions, defining models of the particular columns of the table, is mostly standard for a first-order probabilistic functional language. The expression $D[e_1, \ldots, e_m](F_1, \ldots, F_n)$ represents a draw from a primitive distribution D with hyperparameters determined by the index expressions $e_1, \ldots, e_m$ and parameters defined by the expressions $F_1, \ldots, F_n$. The operator **infer**.$D[e_1, \ldots, e_m].c(E)$ returns an approximate value of the parameter c of the posterior distribution of expression E, expected to be of the form $D[e_1, \ldots, e_m]$. Access to columns defined in previous tables is provided via the operators $t.c$ and $E : t.c$, referencing, respectively, the static attribute with global name c of table t and the E-th row of **inst**-level attribute with global name c of table t. We assume a fixed (but extensible) collection of distributions and deterministic primitives, such as addition, multiplication and comparison.

Distribution signatures are parametrized by spc, to distinguish the use of corresponding distributions in random models and inside queries. The signatures of distributions include the following:

Distributions: $D_{spc} : [x_1 : T_1, \ldots, x_m : T_m](c_1 : U_1, \ldots, c_n : U_n) \to T$

Bernoulli$_{spc}$: (bias : **real**!spc) → **bool**!**rnd**
Beta$_{spc}$ :: (a : **real**!spc, b : **real**!spc) → **real**!**rnd**
Discrete$_{spc}$: [N : **int**!**det**](probs : **real**!spc[N]) → **mod**(N)!**rnd**
Dirichlet$_{spc}$: [N : **int**!**det**](pseudocount : (**real**!spc)[N]) → (**real**!**rnd**)[N]
Gamma$_{spc}$: (shape : **real**!spc, scale : **real**!spc) → **real**!**rnd**
Gaussian$_{spc}$: (mean : **real**!spc, variance : **real**!spc) → **real**!**rnd**
VectorGaussian$_{spc}$:
  [N : **int**!**det**](mean : (**real**!spc)[N], covariance : **real**!spc[N][N]) →
    (**real**!**rnd**[N])

The names of parameters of distributions are fixed and not α-convertible, as they can be referenced by name by the **infer** operator.

Random draws and the **infer** operator were already used in examples in Sections 14.2.2 and 14.2.3. For instance, the expression Dirichlet[2]([1.0, 1.0]) in the column V in table Coins is a random draw from the Dirichlet distribution with the single hyperparameter N (denoting the length of the parameter array and the output array) set to 2 and the single parameter pseudocount (of type (**real**!spc)[2]) set to the array [1.0, 1.0]. Meanwhile, Gaussian([100.0, 100.0]) in the Skill column in table Players is the Gaussian distribution with parameters mean and variance both set to 100. The list of hyperparameters is empty, because (as is clear from the signature) the Gaussian distribution admits no hyperparameters.

The **infer** operator is used in column counts in table Coins in Section 14.2.3. In the expression **infer**.Dirichlet[2].pseudocount(V), V is the name of the column whose posterior distribution we are interested in, Dirichlet[2] is the expected type and hyperparameter vector of that distribution (which we know because of conjugacy) and pseudocount is the name of the parameter of the Dirichlet posterior distribution in column V whose expected value we want to obtain. In other words, if the posterior distribution of the column V returned by the inference algorithm is Dirichlet[2][c_1, c_2], the expression **infer**.Dirichlet[2].pseudocount(V) returns [c_1, c_2], which is the value of the pseudocount parameter of Dirichlet[2][c_1, c_2].

The list of random primitives can be extended by adding multiple signatures for different parametrisations of the same distribution – for instance, the Gaussian

https://doi.org/10.1017/9781108770750.015 Published online by Cambridge University Press

distribution, parametrised above by its mean and variance, can also be parametrised by mean and precision (inverse of variance). This parametrisation is convenient when defining the conjugate Gaussian model.

Distributions: $D_{spc} : [x_1 : T_1, \ldots, x_m : T_m](c_1 : U_1, \ldots, c_n : U_n) \to T$

$\mathsf{GaussianFromMeanAndPrecision}_{spc}$:
$(\mathsf{mean} : \mathbf{real}!spc, \mathsf{prec} : \mathbf{real}!spc) \to \mathbf{real}!\mathbf{rnd}$

In the rest of this chapter, we will abbreviate GaussianFromMeanAndPrecision as GaussianMP.

The syntax of Core Tabular schemas is as follows:

Core Tabular Schemas:

$\mathbb{S} ::= [\,] \mid (t_1 = \mathbb{T}_1) :: \mathbb{S}$	(database) schema
$\mathbb{T} ::= [\,] \mid (c \triangleright x : T\ \ell\ viz\ M) :: \mathbb{T}$	table (or function) (scope of x is $\mathbb{T}$)
$viz ::=$ **input** | **local** | **output**	visibility
$M, N ::= \epsilon \mid E$	model expression

A Tabular *schema* $\mathbb{S}$ consists of any number of named *tables* $\mathbb{T}$, each of which is a sequence of *columns*. Every column in Core Tabular has a *field name* c (also called a *global* or *external name*), an *internal name* x (also called a *local name*), a *type* T (as defined earlier), a *level* (**static** or **inst**), a *visibility* (**input**, **output** or **local**) and a *model expression*, which is empty for **input** columns and is a simple expression E for other types of columns. The **local** visibility is just like **output**, except that **local** columns are not exported to the type of the schema (as defined by the type system, described in Section 14.5), and so can be considered local variables. The default level of a column is **inst**, and we usually omit the level if it is not **static**.

Tables and schemas can also be represented in the formal syntax using list notation. We define $[(c_1 \triangleright x_1 : T_1\ \ell_1\ viz_1\ M_1), \ldots, (c_n \triangleright x_n : T_n\ \ell_n\ viz_n\ M_n)]$ and $[t_1 = \mathbb{T}_1, \ldots, t_n = \mathbb{T}_n]$ to be syntactic sugar for $(c_1 \triangleright x_1 : T_1\ \ell_1\ viz_1\ M_1) :: \ldots :: (c_n \triangleright x_n : T_n\ \ell_n\ viz_n\ M_n) :: [\,]$ and $(t_1 = \mathbb{T}_1) :: \ldots :: (t_n = \mathbb{T}_n) :: [\,]$, respectively.

To illustrate the formal syntax of Tabular, let us consider again the simple Coins table, which was presented in the grid-based form at the beginning of Section 14.2.2. If we specify the global and local names explicitly, this table has the following form:

table Coins			
V ▹ V	**real**!**rnd**[2]	**static output**	Dirichlet[2]([1.0, 1.0])
Flip ▹ Flip	**mod**(2)!**rnd**	**output**	Discrete[2](V)

https://doi.org/10.1017/9781108770750.015 Published online by Cambridge University Press

In the formal Tabular syntax, this table would be written as follows:

$$\begin{aligned}[&(\mathsf{V} \triangleright \mathsf{V} : \textbf{real}!\textbf{rnd}[2]\ \textbf{static output}\ \mathsf{Dirichlet}[2]([1.0, 1.0]),\\ &\quad(\mathsf{Flip} \triangleright \mathsf{Flip} : \textbf{mod}(2)!\textbf{rnd inst output}\ \mathsf{Discrete}[2](V))]\end{aligned}$$

The schema consisting of just this single table is:

$$\begin{aligned}[\mathsf{Coins} = [&(\mathsf{V} \triangleright \mathsf{V} : \textbf{real}!\textbf{rnd}[2]\ \textbf{static output}\ \mathsf{Dirichlet}[2]([1.0, 1.0]),\\ &\quad(\mathsf{Flip} \triangleright \mathsf{Flip} : \textbf{mod}(2)!\textbf{rnd inst output}\ \mathsf{Discrete}[2](V))]]\end{aligned}$$

In the rest of this chapter, col denotes a single column $(c \triangleright x : T\ \ell\ viz\ M)$ of a table, where its components are unimportant.

Motivation for double column names In the syntax of the new version of Tabular presented in the paper which was the starting point for this work (Gordon et al., 2015), each column only has one name. This causes a problem with alpha-conversion: if a column is visible outside the given table, then its name cannot be alpha-convertible, since renaming the column would break references to it from outside the table. On the other hand, alpha-conversion is necessary for the substitution and function reduction to work properly. To mitigate this issue, we now follow the standard approach used in module systems, first presented by Harper and Lillibridge (1994): we give each column two names, a local, alpha-convertible name, which is only in scope of a given table, and a global, fixed field name, which can only be used outside the table (or function). In practice, we can assume that the internal and external name are initially the same.

14.3.3 Syntax of Schemas with Functions and Indexing

Tabular supports two additional kinds of model expressions: *function applications* and *indexed models*.

A function is represented as a Core table whose last column is identified by the name ret and has visibility **output**. A function $\mathbb{T}$ can be applied to a list of named *arguments* R, whose types and number must match the types and number of **input** columns in the function table. Note that function arguments are identified by the field name of the corresponding column. The reduction algorithm (presented in Section 14.4) reduces a column containing a function application to the body of the function with all **input** columns removed and the input variables in subsequent model expressions replaced by the corresponding arguments.

The output column of a function can be referenced in the "caller" table simply by the (local) name of the "caller" column. Other columns can be referenced by means of a new operator $e.c$, where e is expected to be the local name x of the

https://doi.org/10.1017/9781108770750.015 Published online by Cambridge University Press

"caller" column and c is the field name of the referenced column of the function table (we need to use the field name, because the local name is only in scope inside the function).

An *indexed model* $M[e_{index} < e_{size}]$ represents the model M with all **rnd static** attributes turned into arrays of size e_{size} and references to them replaced by array lookups extracting the element at index e_{index}.

Full Tabular Schemas:

$E ::= \cdots \mid e.c$	expression
$M, N ::= \cdots \mid M[e_{index} < e_{size}] \mid \mathbb{T}\ R$	model expression
$R ::= [] \mid (c = e) :: R$	function arguments

Function arguments can also be represented using the standard list notation as $R = (c_1 = e_1, \ldots, c_n = e_n)$. The function field reference is only defined to be $e.c$ rather than $x.c$ in order for substitution to be well-defined. The indexing operator is only meaningful if it is applied (possibly multiple times) to a function application, since it has no effect on basic expressions.

In the Coins example in Section 14.2.2, a predefined function (from the standard library) was referenced in the main table by its name. Indeed, in the implementation, functions are always defined outside of the main schema and are called by identifiers. In the formal syntax, however, functions are inlined. For instance, to represent the function call CDiscrete(N = 2, R = 1.0) formally, we need to substitute CDiscrete with its body. The resulting function application looks as follows:

$$
\begin{array}{l}
[(\mathsf{N} \triangleright \mathsf{N} : \textbf{int}!\textbf{det static input}\ \epsilon),\\
\ (\mathsf{R} \triangleright \mathsf{R} : \textbf{real}!\textbf{det static input}\ \epsilon),\\
\ (\mathsf{V} \triangleright \mathsf{V} : \textbf{real}!\textbf{rnd}[\mathsf{N}]\ \textbf{static input}\ \mathsf{Dirichlet}[\mathsf{N}]([\textbf{for}\ \mathsf{i} < \mathsf{N} \rightarrow R])),\\
\ (\mathsf{ret} \triangleright \mathsf{ret} : \textbf{mod}(\mathsf{N})!\textbf{rnd}[\mathsf{N}]\ \textbf{static input}\ \mathsf{Dirichlet}[\mathsf{N}]([\textbf{for}\ \mathsf{i} < \mathsf{N} \rightarrow R]))]\\
\qquad (\mathsf{N} = 2, \mathsf{R} = 1)
\end{array}
$$

Free Variables and Core *Columns*

The free variables $\mathrm{fv}(\mathbb{T})$ of a table $\mathbb{T}$ are all local variables used in column types and model expressions which are not bound by column declarations or for-loops. Formally, the operator $\mathrm{fv}(\mathbb{T})$ can be defined inductively in the usual way. Unbound occurrences of field names are not considered free variables, as they are a separate syntactic category.

The predicate Core states that the given schema, table or column is in Core form, as defined earlier.

14.4 Reduction to Core Tabular

We now define the reduction relation which reduces arbitrary well-typed Tabular schemas (with function applications and indexing) to a Core form. Before discussing the technical details of reduction, we present an example which will guide our development. This time we make the distinction between local and field names explicit, to illustrate how substitution and renaming work.

Consider the following function implementing the widely used Conjugate Gaussian model, whose output is drawn from a Gaussian with mean modelled by another Gaussian and precision (inverse of variance) drawn from a Gamma distribution:

fun CG			
M ▹ M	**real!det**	**static input**	
P ▹ P	**real!det**	**static input**	
Mean ▹ Mean	**real!rnd**	**static output**	GaussianMP(M,P)
Prec ▹ Prec	**real!rnd**	**static output**	Gamma(1.0, 1.0)
ret ▹ ret	**real!rnd**	**output**	GaussianMP(Mean, Prec)

Suppose we want to use this function to model eruptions of the Old Faithful geyser in the Yellowstone National Park. The eruptions of this geyser, known for its regularity, can be split into two clusters based on their duration and time elapsed since the previous eruption: some eruptions are shorter and occur more frequently, others are longer but one has to wait longer to see them. Given a database consisting of eruption durations and waiting times (not split into clusters), we want to infer the means and precisions of the distributions of durations and waiting times in each of the two clusters. If we simply modelled the duration and waiting time with a call to CG, we would obtain a single distribution for the mean and precision of each quantity, but we can turn each Mean and Prec column into an array of size 2 by combining the function calls with indexing.

table Faithful			
cluster ▹ cluster	**mod**(2)!**rnd**	**output**	(CDiscrete(N=2)
duration ▹ duration	**real!rnd**	**output**	CG(M=0.0, P=1.0)[cluster<2]
time ▹ time	**real!rnd**	**output**	CG(M=60.0, P=1.0)[cluster<2]

14.4.1 Reducing Function Applications

Before we introduce the reduction of indexed models, let us consider a simplified version of the above model, with just function applications:

table Faithful			
duration ▹ duration	**real!rnd**	**output**	CG(M=0.0, P=1.0)
time ▹ time	**real!rnd**	**output**	CG(M=60.0, P=1.0)

To reduce the duration and time columns to Core form, we must expand the applications. This is done by just replacing the given column with the body of the function with the arguments substituted for the input variables. The field name of the last column, always expected to be the keyword ret, is replaced by the name of the “caller” column, and the field names of previous columns are prefixed with the field name of the “caller” column. This is done to ensure that field names in the reduced table are unique, even if the same function is used several times.

Meanwhile, local names can be refreshed (by alpha-conversion), to make sure they do not clash with variables which are free in the remainder of the “caller” table or the remaining arguments. References to the columns of the function in the “caller” table (of the form $x.c$) are then replaced with the refreshed local column names.

In the end, the above table reduces to the following form:

table Faithful			
duration.Mean ▹ Mean	**real!rnd**	**static output**	GaussianMP(0.0,1.0)
duration.Prec ▹ Prec	**real!rnd**	**static output**	Gamma(1.0,1.0)
duration ▹ duration	**real!rnd**	**output**	GaussianMP(Mean,Prec)
time.Mean ▹ Mean	**real!rnd**	**static output**	GaussianMP(60.0,1.0)
time.Prec ▹ Prec	**real!rnd**	**static output**	Gamma(1.0,1.0)
time ▹ time	**real!rnd**	**output**	GaussianMP(Mean,Prec)

Just like in ordinary languages, variable definitions can be overshadowed by more closely scoped binders. The variable Mean in the duration column refers to the definition in the column with external name duration.mean, and Mean in column time refers to the definition in the column with field name time.Mean, and similarly with Prec.

Binders and Capture-avoiding Substitutions: $\mathbb{T}\{^{e}/_{x}\}$, $\mathbb{T}\langle ^{y}/_{x.c}\rangle$

In order to define the reduction rules, we first need two capture-avoiding substitution operators on tables: $\mathbb{T}\{^{e}/_{x}\}$, which replaces free occurrences of the variable x with the index expression e, and $\mathbb{T}\langle ^{y}/_{x.c}\rangle$, which replaces function field references $x.c$ with a single local variable y. These substitutions can be formally defined inductively, as usual. Here we omit these formal definitions (which can be found in Szymczak, 2018) and show by example how the second, slightly less standard, operator works.

Let us consider again the simplified version of the Old Faithful model from the beginning of this section, but this time using different local variable and field names, to emphasise the fact that they are not the same thing:

table Faithful			
duration ▹ x	**real!rnd**	**output**	CG(M=0.0, P=1.0)
time ▹ x’	**real!rnd**	**output**	CG(M=60.0, P=1.0)

Suppose we want to calculate the mean of the posterior distribution of the mean of duration (using the **infer** operator, described in 14.2.3). To this end, we need to add an additional column to the above table, which references the column with field name Mean in the reduced application of CG in the column duration. As field names are not binders, we need to use the local name x of the column duration. On the other hand, as the local names of the columns of CG are not visible outside the

https://doi.org/10.1017/9781108770750.015 Published online by Cambridge University Press

function CG itself, we need to access the column Mean of CG by using its field name. Hence, the reference has the form x.Mean, and the full table is the following:

table Faithful			
duration ▹ x	**real!rnd**	**output**	CG(M=0.0, P=1.0)
time ▹ x’	**real!rnd**	**output**	CG(M=60.0, P=1.0)
duration_mean ▹ z	**real!qry**	**output**	**infer**.Gaussian.mean(x.Mean)

When the function application in column duration is reduced (as described later), and the column Mean of the application of CG in duration is turned into a column with local name y in the main table, we need to substitute references to the (no longer existing) parameter Mean of the model in column x with the variable y in the rest of the table by using the operator $\langle {}^{y}\!/_{x.c} \rangle$. Applying this substitution to the last two columns of the above table yields:

time ▹ x’	**real!rnd**	**output**	CG(M=60.0, P=1.0)
duration_mean ▹ z	**real!qry**	**output**	**infer**.Gaussian.mean(y)

One might be concerned that the substitution $\langle {}^{y}\!/_{x.c} \rangle$ would not work correctly if the function application pointed to by x was assigned to another variable z, for example in a part of a table of the form:

field1 ▹ z	**real!rnd**	**output**	x
field2 ▹ z’	**real!rnd**	**output**	z.c

However, it is impossible to assign a function application to another variable in Tabular, as it is impossible to reference a function application as a whole. If a variable x referencing a function application is used on its own (not in a field reference $x.c$), it always denotes the *last column* of the reduced application, not the application itself. The expression $z.c$ in the above table is not well-typed, as z does not refer to a function.

Reduction Relation

The reduction is defined by means of the small-step reduction relation, reducing one column of the function table at a time, being the least relation closed under the set of rules presented below.

Reduction to Core Tabular:

$\mathbb{T} \to \mathbb{T}'$ table reduction

The judgment $\mathbb{T} \to \mathbb{T}'$ states that table $\mathbb{T}$ reduces to $\mathbb{T}'$ in one step. In the reduction rules, we normally use o for the (field) name of the “caller” column and c for the name of a column in the function table, to disambiguate between the two. The reduction system is deterministic and the assumptions guarantee that at most one rule applies to each table (the same applies to the reduction rules for indexed models and schemas presented in the following sections).

Reduction Rules for Tables: $\mathbb{T} \to \mathbb{T}'$

(Red Appl Output) (for Core($\mathbb{T}$))

$$\frac{y \notin \mathrm{fv}(\mathbb{T}', R) \cup \{x\} \qquad c \neq \mathsf{ret}}{\begin{array}{l}(o \triangleright x : T\ \ell\ \mathit{viz}\ ((c \triangleright y : T'\ \ell'\ \textsf{\textbf{output}}\ E) :: \ \mathbb{T})\ R) :: \mathbb{T}' \to \\ \quad (o.c \triangleright y : T'\ (\ell \wedge \ell')\ \mathit{viz}\ E) :: (o \triangleright x : T\ \ell\ \mathit{viz}\ \mathbb{T}\ R) :: \mathbb{T}'\langle {}^{y}\!/\!_{x.c}\rangle\end{array}}$$

(Red Appl Local) (for Core($\mathbb{T}$))

$$\frac{y \notin \mathrm{fv}(\mathbb{T}', R) \cup \{x\}}{\begin{array}{l}(o \triangleright x : T\ \ell\ \mathit{viz}\ ((c \triangleright y : T'\ \ell'\ \textsf{\textbf{local}}\ E) :: \mathbb{T})\ R) :: \mathbb{T}' \to \\ \quad (\_ \triangleright y : T'\ (\ell \wedge \ell')\ \textsf{\textbf{local}}\ E) :: (o \triangleright x : T\ \ell\ \mathit{viz}\ \mathbb{T}\ R) :: \mathbb{T}'\end{array}}$$

(Red Appl Input) (for Core($\mathbb{T}$))

$$\frac{}{\begin{array}{l}(o \triangleright x : T\ \ell\ \mathit{viz}\ (c \triangleright y : T'\ \ell'\ \textsf{\textbf{input}}\ \epsilon) :: \mathbb{T}\ (c = e) :: R) :: \mathbb{T}' \to \\ \quad (o \triangleright x : T\ \ell\ \mathit{viz}\ \mathbb{T}\left\{{}^{e}\!/\!_{y}\right\}\ R) :: \mathbb{T}'\end{array}}$$

(Red Appl Ret)

$$\frac{}{\begin{array}{l}(o \triangleright x : T\ \ell\ \mathit{viz}\ [(\mathsf{ret} \triangleright y : T'\ \ell'\ \textsf{\textbf{output}}\ E)]\ []) :: \mathbb{T}' \to \\ \quad (o \triangleright x : T'\ (\ell \wedge \ell')\ \mathit{viz}\ E) :: \mathbb{T}'\end{array}}$$

(Red Table Right)

$$\frac{\mathbb{T} \to \mathbb{T}' \qquad \mathsf{Core}(\mathsf{col})}{\mathsf{col} :: \mathbb{T} \to \mathsf{col} :: \mathbb{T}'}$$

The (Red Appl Output) rule (in which *viz* is expected to be **local** or **output**) reduces a single **output** column of a function by appending it to the main table, preceded by the "caller" column with the unevaluated part of the application $\mathbb{T}\ R$ (which will be reduced in the following steps). If the function was called from a **static** column, the level of the reduced function column is changed to **static**. Similarly, if the function was called from a **local** column, the visibility of the reduced column is dropped to **local**. Because the reduced column is appended to the main table, it has to be referenced using its internal name (recall that field names are not binders). Hence, all references to it, of the form $x.c$, are replaced with its internal name y. Meanwhile, the global name of the reduced column is prefixed by the field name of the "caller" column.

To avoid capturing free variables which are not bound by the reduced column in the original top-level table, y is required not to be free in $\mathbb{T}'$ and R. This is always possible, because tables are identified up to alpha-conversion of internal column

https://doi.org/10.1017/9781108770750.015 Published online by Cambridge University Press

names, so y can be refreshed if needed (formally, the reduction relation is a relation on alpha-equivalence classes of syntactic terms).

(Red Appl Local) is similar, except that we do not need to substitute y for $x.c$ in $\mathbb{T}$, because the given column is not visible outside the function. The external name of a reduced column can be empty, because local columns are not exported.

The (Red Appl Input) rule removes an input column and replaces all references to it in the rest of the function with the corresponding argument.

The last column of a function is reduced by (Red Appl Ret), which simply replaces the application of the single ret column to the empty argument list with the expression from the said column. The level is also changed to **static** if the ret column was **static**. The internal and field names of the top-level column are left unchanged, and the names of the last column of the function are discarded, because the last column of a function is always referenced by the name of the "caller" table.

(Red Table Right) is a congruence rule, allowing us to move to the next column of the main table if the current first column is already in Core form.

Example of Function Reduction To see how the reduction rules work, let us consider again the simplified version of the Old Faithful example with the additional duration_mean column:

table Faithful			
duration ▹ x	**real!rnd**	**output**	CG(M=0.0, P=1.0)
time ▹ x'	**real!rnd**	**output**	CG(M=60.0, P=1.0)
duration_mean ▹ z	**real!qry**	**output**	**infer**.Gaussian.mean(x.Mean)

The reduction rules reduce the duration column first. In the beginning, the rule (Red Appl Input) is applied twice, and reduces the columns M and P of the function CG in duration, replacing references to M and P in the body of CG with corresponding arguments. The reduced table has the following form:

table Faithful			
duration ▹ x	**real!rnd**	**output**	CG'()
time ▹ x'	**real!rnd**	**output**	CG(M=60.0, P=1.0)
duration_mean ▹ z	**real!qry**	**output**	**infer**.Gaussian.mean(x.Mean)

where CG' is the following partially evaluated function:

fun CG'			
Mean ▹ Mean	**real!rnd**	**static output**	GaussianMP(0.0,1.0)
Prec ▹ Prec	**real!rnd**	**static output**	Gamma(1.0, 1.0)
ret ▹ ret	**real!rnd**	**output**	GaussianMP(Mean, Prec)

The next rule to be applied is (Red Appl Output), which reduces the first column Mean of CG' and replaces references to it, of the form x.Mean, with the local name of the reduced column (which we can assume is still Mean, as the name does not conflict with any other variable), in the rest of the top-level table by using the field substitution operator. The reduced table has the following form:

https://doi.org/10.1017/9781108770750.015 Published online by Cambridge University Press

table Faithful

duration.Mean ▹ Mean	**real!rnd**	**static output**	GaussianMP(0.0,1.0)
duration ▹ x	**real!rnd**	**output**	CG”()
time ▹ x’	**real!rnd**	**output**	CG(M=60.0, P=1.0)
duration_mean ▹ z	**real!qry**	**output**	**infer**.Gaussian.mean(Mean)

where CG’’ is:

fun CG”

Prec ▹ Prec	**real!rnd**	**static output**	Gamma(1.0, 1.0)
ret ▹ ret	**real!rnd**	**output**	GaussianMP(Mean, Prec)

Note that Mean in CG’’ refers to the column defined outside the function (which is in scope of CG’’, as functions are assumed to be defined inline, even though the implementation uses named functions).

The remaining columns of function applications are reduced similarly, except that the local name Mean in the second application of CG has to be changed by α-conversion, as Name is free in the last column of the top-level table.

14.4.2 Reducing Indexed Models

In order to reduce a column with an indexed function application, we need to transform the function into an indexed form before applying it to the arguments. In the case of the duration column of the original table of the running example, this transformation needs to turn the expressions in all **static rnd** columns into arrays of size 2, with each element modelled by the original expression, and replace all references to these columns in the rest of the table with array accesses, returning the component at index cluster.

For instance, applying indexing [cluster $<$ 2] to the function CG yields the following indexed function

M ▹ M	**real!det**	**static input**	
P ▹ P	**real!det**	**static input**	
Mean ▹ Mean	**real!rnd**	**static output**	[**for** _ < 2 → GaussianMP(M,P)]
Prec ▹ Prec	**real!rnd**	**static output**	[**for** _ < 2 → Gamma(1.0, 1.0)]
ret ▹ ret	**real!rnd**	**output**	GaussianMP(Mean[cluster], Prec[cluster])

parametrised on the free variable cluster defined outside the function.

Reducing the application of this function to (M $=$ 0.0, P $=$ 1.0) in the duration column gives the following table:

duration.Mean ▹ Mean	**real!rnd**[2]	**static output**	[**for** _ < 2 → GaussianMP(0.0,1.0)]
duration.Prec ▹ Prec	**real!rnd**[2]	**static output**	[**for** _ < 2 → Gamma(1.0,1.0)]
duration ▹ duration	**real!rnd**	**output**	GaussianMP (Mean[cluster], Prec[cluster)

More generally, table indexing is formalised via the operator $\text{index}_A(\mathbb{T}, e_1, e_2)$, where $\mathbb{T}$ is the table (reduced application) to index, e_1 and e_2 are, respectively, the index variable and the number of clusters and A is the (initially empty) set of **static**

rnd columns, which needs to be available to convert variables into array accesses correctly.

We disallow indexing tables with **qry** columns, since substituting a reference to a query column with an array access with a random index would break the information flow constraints, so indexed query columns would not have a well-defined semantics. Below, the predicate NoQry states that a given Core table or model has no **qry**-level columns. The requirement that tables with **qry** columns cannot be indexed is enforced by the type system, presented in Section 14.5.

The indexing operator makes use of a new capture-avoiding substitution operator: $E[A, e]$ denotes E with every variable x in the set of variables A (supposed to contain only **static rnd** variables) replaced with the array access $x[e]$, as long as the syntax allows it. For instance, $\mathsf{Gaussian}(x, y)[\{x, y\}, i]$ is $\mathsf{Gaussian}(x[i], y[i])$, but $(\mathsf{Discrete}[z](y))[\{z, y\}, i]$ is $\mathsf{Discrete}[z](y[i])$, and not $\mathsf{Discrete}[z[i]](y[i])$, because hyperparameters of distributions are index expressions, so they cannot be array accesses. However, we do not need to worry about variables which cannot be replaced with array accesses, such as z above, as (in non-**qry** columns of functions) they are always expected to be deterministic or occur in function field references of the form $x.c$, while indexing is only supposed to modify random variables referencing Core columns. We elide the formal definition of the operator, which can be found in Szymczak (2018).

The indexing operator is defined inductively below.

Table Indexing: $\mathrm{index}_A(\mathbb{T}, e_1, e_2)$**, where** $\mathsf{NoQry}(\mathbb{T})$

$$\mathrm{index}_A([], e_1, e_2) \triangleq []$$

$$\begin{aligned}&\mathrm{index}_A((c \triangleright x : T\ \textsf{\textbf{static}}\ viz\ E) :: \mathbb{T}, e_1, e_2) \triangleq\\ &\quad (c \triangleright x : T[e_2]\ \textsf{\textbf{static}}\ viz\ [\textsf{\textbf{for}}\ i < e_2 \rightarrow E[A, i]]) :: \mathrm{index}_{A \cup \{x\}}(\mathbb{T}, e_1, e_2)\\ &\qquad \text{if } viz \neq \textsf{\textbf{input}} \text{ and } \textsf{\textbf{rnd}}(T) \text{ and } x \notin \mathrm{fv}(e_1) \cup \mathrm{fv}(e_2) \cup A \text{ and } i \notin \mathrm{fv}(E)\end{aligned}$$

$$\begin{aligned}&\mathrm{index}_A((c \triangleright x : T\ \ell\ \textsf{\textbf{input}}\ \epsilon) :: \mathbb{T}, e_1, e_2) \triangleq\\ &\quad (c \triangleright x : T\ \ell\ \textsf{\textbf{input}}\ \epsilon) :: \mathrm{index}_A(\mathbb{T}, e_1, e_2) \text{ if } x \notin \mathrm{fv}(e_1) \cup \mathrm{fv}(e_2) \cup A\end{aligned}$$

$$\begin{aligned}&\mathrm{index}_A((c \triangleright x : T\ \ell\ viz\ E) :: \mathbb{T}, e_1, e_2) \triangleq\\ &\quad (c \triangleright x : T\ \ell\ viz\ E[A, e_1]) :: \mathrm{index}_A(\mathbb{T}, e_1, e_2)\\ &\qquad \text{otherwise if } x \notin \mathrm{fv}(e_1) \cup \mathrm{fv}(e_2) \cup A.\end{aligned}$$

Unsurprisingly, indexing an empty table returns an empty table. In any **static rnd** column, the model expression E is turned into an array of e_2 elements, each modelled by E. Since E may contain references to previous **static rnd** columns of the original table, which have been turned into arrays, we must replace these references (by means of the $E[A, i]$ operator) with array accesses, returning values at indices corresponding to the positions of the expressions. Before index is applied recursively to the rest of the table, the variable x is added to the set A of **rnd static** variables, so that each reference to x in subsequent **rnd static** and **rnd inst** columns would be replaced with an appropriate array access.

Input columns are left unchanged by index, and in **inst**-level random columns, references to previous **static rnd** columns are replaced by array accesses returning the e_1 -th component. Note that $E[A,i]$ leaves expressions in deterministic columns unchanged, because all variables in the set A are expected to be random.

With the index operator in place, we can define the reduction relation reducing indexed models.

Reduction to Core Tabular:

$M \rightarrow M'$ model reduction

The above judgment, which states that indexed model M reduces to M' in one step (that is, that M' is M with one level of indexing eliminated), is derived by the following rules:

Reduction Rules for Models: $M \rightarrow M'$

(Red Index)

$$\frac{\mathsf{Core}(\mathbb{T}) \quad \mathsf{NoQry}(\mathbb{T})}{(\mathbb{T}\ R)[e_{index} < e_{size}] \rightarrow (\mathsf{index}_{\varnothing}(\mathbb{T}, e_{index}, e_{size}))\ R}$$

(Red Index Inner)

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

(Red Index Expr)

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

Reduction Rules for Tables: $\mathbb{T} \rightarrow \mathbb{T}'$

(Red Model)

$$\frac{M \rightarrow M'}{(c \triangleright x : T\ \ell\ viz\ M) :: \mathbb{T} \rightarrow (c \triangleright x : T\ \ell\ viz\ M') :: \mathbb{T}}$$

The (Red Index) rule applies the index operator to the function table in an application, returning a pure function application which will be reduced at table level. The (Red Index Inner) rule simply allows reducing a model nested in an indexed expression, in case this model is an indexed model itself. Since simple expressions have no static parameters of their own, indexing a simple expression has no effect, so the (Red Index Expr) rule just discards the indexing. The (Red Model) rule allows reducing a model (other than a function application) in a column of a table.

14.4.3 Reducing Schemas

Finally, we can define the reduction relation for schemas:

Reduction to Core Tabular:

$\mathbb{S} \rightarrow \mathbb{S}'$	schema reduction

The judgment $\mathbb{S} \rightarrow \mathbb{S}'$ states that the schema $\mathbb{S}$ reduces to $\mathbb{S}'$ in one step – that is, $\mathbb{S}'$ is $\mathbb{S}$ with one table reduced to Core form. This judgment is derived by the following two rules:

Reduction Rules for Schemas: $\mathbb{S} \rightarrow \mathbb{S}'$

(Red Schema Left)

$$\frac{\mathbb{T} \rightarrow \mathbb{T}'}{(t = \mathbb{T}) :: \mathbb{S} \rightarrow (t = \mathbb{T}') :: \mathbb{S}}$$

(Red Schema Right)

$$\frac{\mathbb{S} \rightarrow \mathbb{S}' \quad \mathsf{Core}(\mathbb{T})}{(t = \mathbb{T}) :: \mathbb{S} \rightarrow (t = \mathbb{T}) :: \mathbb{S}'}$$

The (Red Schema Left) rule reduces the first table, while (Red Schema Right) proceeds to the following table if the first one has already been fully reduced.

Putting all these rules together, we can finally reduce the Old Faithful model to Core form:

table faithful			
cluster.V ▹ V	**real**!**rnd**[2]	**static output**	Dirichlet[2]([**for** i < 2 → 1.0])
cluster ▹ cluster	**mod**(2)!**rnd**	**output**	Discrete[2](V)
duration.Mean ▹ Mean	**real**!**rnd**[2]	**static output**	[**for** i < 2 → GaussianMP(0.0, 1.0)]
duration.Prec ▹ Prec	**real**!**rnd**[2]	**static output**	[**for** i < 2 → Gamma(1.0, 1.0)]
duration ▹ duration	**real**!**rnd**	**output**	GaussianMP(Mean[cluster], Prec[cluster])
time.Mean ▹ Mean	**real**!**rnd**[2]	**static output**	[**for** i < 2 → GaussianMP(60.0, 1.0)]
time.Prec ▹ Prec	**real**!**rnd**[2]	**static output**	[**for** i < 2 → Gamma(1.0, 1.0)]
time ▹ time	**real**!**rnd**	**output**	GaussianMP(Mean[cluster], Prec[cluster])

As noted before, a Tabular model in Core form has a straightforward interpretation as a factor graph. Assuming that the table faithful has n rows, the reduced Old Faithful model corresponds to the (directed) factor graph shown in Figure 14.1, in which we use abbreviated variable names (for example dM for duration.Mean) to make the presentation cleaner:

The boxes with solid edges are *plates*, which create multiple copies of given variables and factors – for instance, we have n values of dM_i, one for each i, each drawn from the same distribution GaussianMP$(0.0, 1.0)$. The boxes with dotted lines are *gates* (Minka and Winn, 2009), which select a factor based on the value of a categorical variable (c_j in this case). While the graph above is directed to make the dependency structure explicit, the arrow heads can be removed to obtain a standard, undirected factor graph.

https://doi.org/10.1017/9781108770750.015 Published online by Cambridge University Press

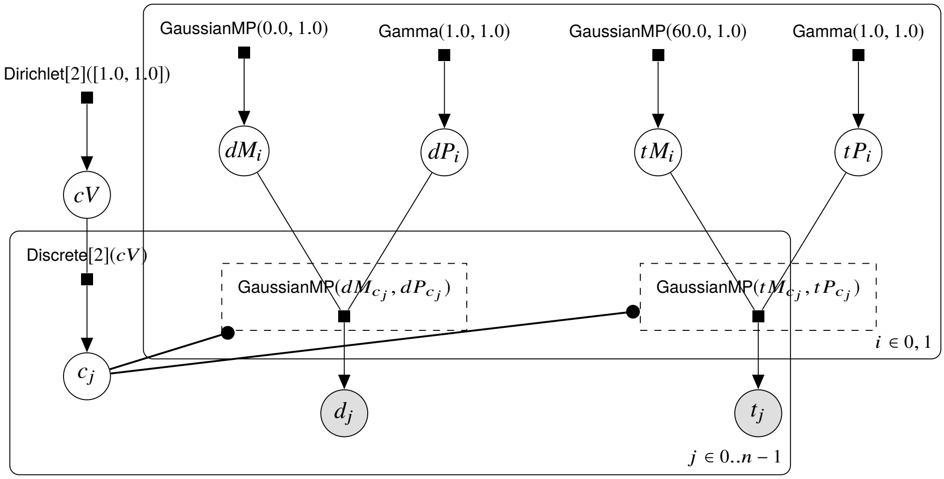


Figure 14.1 Reduced Old Faithful model as a factor graph

Once a schema is reduced to Core form, the Tabular backend can perform inference on it using Expectation Propagation or another algorithm provided by Infer.NET. Figure 14.2 presents the results of inference shown as in the Excel interface and visualised by Excel plots. A new column assignment has been added to the model – this column uses the **infer** operator to assign each eruption to the more likely cluster.

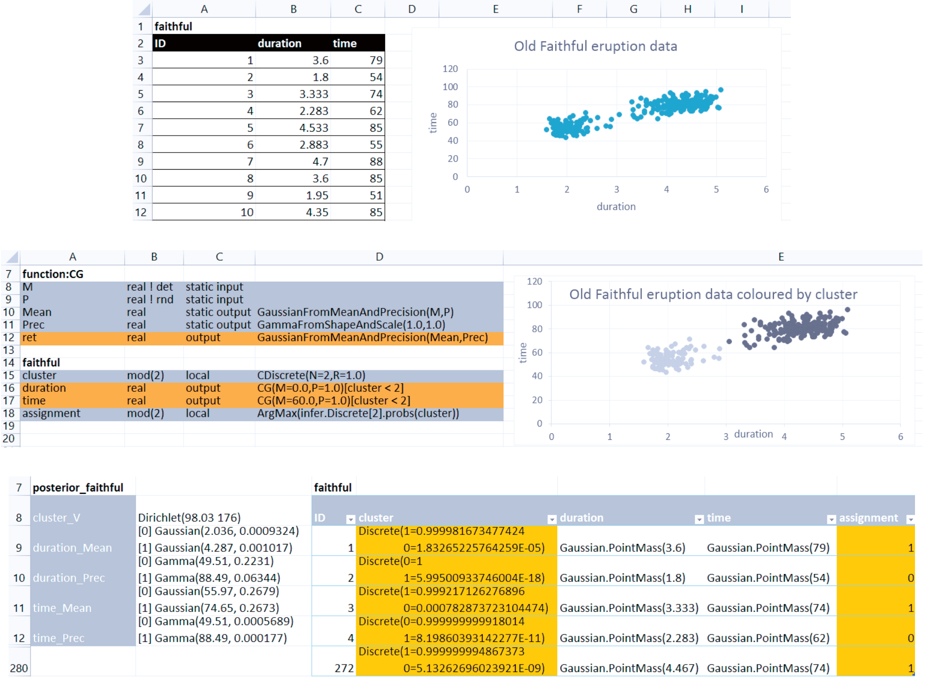
14.5 Type System

Type systems are useful in probabilistic languages because they specify the domain of each random variable and ensure that each random draw is used where a value in the given domain is expected. Thus, types guide the modelling process and help prevent incorrect dependencies between variables.

As seen in examples in the previous sections, Tabular makes use of basic dependent types and determinacy and binding time annotations. All the type constraints in Tabular are checked statically, which allows some modelling errors to be caught before the inference procedure is started, thus saving the user time on debugging.

Tabular's type system ensures that the role of each variable in the program is immediately clear and checks that each random variable is defined on the right domain. Dependent types additionally allow checking array sizes and bounds on categorical variables in functions, even though these may depend on function arguments. This helps to check that functions are indeed correctly defined and make the right use of their arguments. Moreover, because of space annotations, the compiler

https://doi.org/10.1017/9781108770750.015 Published online by Cambridge University Press



	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**	
8	cluster_V	Dirichlet(98.03 176)
9	duration_Mean	[0] Gaussian(2.036, 0.0009324) [1] Gaussian(4.287, 0.001017)
10	duration_Prec	[0] Gamma(49.51, 0.2231) [1] Gamma(88.49, 0.06344)
11	time_Mean	[0] Gaussian(55.97, 0.2679) [1] Gaussian(74.65, 0.2673)
12	time_Prec	[0] Gamma(49.51, 0.0005689) [1] Gamma(88.49, 0.000177)
280		

faithful

ID	cluster	duration	time	assignment
1	Discrete(1=0.999981673477424 0=1.83265225764259E-05)	Gaussian.PointMass(3.6)	Gaussian.PointMass(79)	1
2	Discrete(0=1 1=5.99500933746004E-18)	Gaussian.PointMass(1.8)	Gaussian.PointMass(54)	0
3	Discrete(1=0.999217126276896 0=0.000782873723104474)	Gaussian.PointMass(3.333)	Gaussian.PointMass(74)	1
4	Discrete(0=0.999999999918014 1=8.19860393142277E-11)	Gaussian.PointMass(2.283)	Gaussian.PointMass(62)	0
272	Discrete(1=0.999999994867373 0=5.13262696023921E-09)	Gaussian.PointMass(4.467)	Gaussian.PointMass(74)	1

Figure 14.2 Old Faithful inference results in Excel

can split a Tabular program into the probabilistic model and the post-processing code, and process them at the right time in the pipeline – the user is not required to define **rnd** and **qry** variables in separate blocks. These annotations also disallow models which the default inference engine cannot handle, such as mixture models with an random number of components.

In this section, we define the Tabular type system formally and present the type soundness property of the reduction system shown in Section 14.4 (whose proof can be found in Szymczak (2018)).

In addition to the column types introduced in Section 14.3, we also give types to model expressions, tables and schemas. These types define the spaces of input and output variables of the probabilistic models defined by programs or their parts.

Limitations of the Type System The type system does not enforce conjugacy, which is required by the default inference engine of Tabular, because we wanted to keep the developments in this chapter independent of a particular inference algorithm. Moreover, well-typedness of a Tabular program does not guarantee that Expectation Propagation inference will always succeed. Lack of conjugacy and other algorithm-specific issues may result in the inference algorithm failing at

https://doi.org/10.1017/9781108770750.015 Published online by Cambridge University Press

runtime, in which case an error message from the inference backend is shown to the user in the implementation.

14.5.1 Syntax of Tabular Types

To each model and table, we assign a type Q (hereafter called Q-type), which consists of a list of column names (local and field names), column types, levels and visibilities (which cannot be **local**, because local attributes of tables and functions are not exported to types). A single component of type Q is just a table column without a model expression. The Q-types used here are akin to right-associating dependent record types (Pollack, 2002), except that in their inhabitants, the values of fields may depend on previous fields, like in translucent sums (Harper and Lillibridge, 1994).

The type Sty of a schema is just a list of table identifiers paired with corresponding table types. These types are notably simpler than the nested record types used in the original formulation of Tabular (Gordon et al., 2014b).

We define three predicates on Q-types: **fun**(Q), which means that the given type Q is a valid function type, whose last column is marked as the return column, **table**(Q), which states that Q has no deterministic **static** columns and can type a top-level (i.e. non-function) table, and red(Q), which states that Q is the type of a reduced function application, having no **input** columns.

Table and Schema Types:

$Q ::= [\,] \mid (c \triangleright x : T\ \ell\ viz) :: Q$	table type (scope of x is Q, $viz \neq$ **local**)
$Sty ::= (t : Q) :: Sty$	schema type

fun(Q) iff $viz_n =$ **output** and $c_n =$ ret

table(Q) iff for each $i \in 1..n$, $\ell_i =$ **static** $\Rightarrow$ **rnd**(T_i) $\vee$ **qry**(T_i)

red(Q) iff **table**(Q) and for each $i \in 1..n$, $viz_i =$ **output**

The predicate **table**(Q) ensures that no top-level columns can be referenced in subsequent column types (because only **static det** columns can appear in types), which guarantees that all column types in Core tables (including reduced tables) are closed, except possibly for table size references. This property is necessary because columns can be referenced from other tables, and any variables in a type would be free outside the table in which the corresponding column was defined.

We define fv(Q) to be the set of local variables in column types in Q which are not bound by column definitions.

Schemas, tables, models and expressions are all typechecked in a given typing environment Γ, which is an ordinary typing environment except that it has three kinds of entries (for variables denoting previous Core columns, for previous tables

https://doi.org/10.1017/9781108770750.015 Published online by Cambridge University Press

and for reduced function applications) and the entries for Core columns include level annotations as well as column types (recall that column types themselves contain binding type annotations).

Tabular Typing Environments:

$\Gamma ::= \varnothing \mid (\Gamma, x :^{\ell} T) \mid (\Gamma, t : Q) \mid (\Gamma, x : Q)$	environment

The *domain* dom(Γ) of an environment Γ is the set of all variables and table names in the environment:

Below is the list of all judgments of the Tabular type system, which will be described in the remainder of this section.

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
$\Gamma \vdash^{pc} E : T$	in Γ at level pc, expression E has type T
$\Gamma \vdash^{pc} R : Q \to Q'$	R sends function type Q to model type Q'
$\Gamma \vdash^{pc} M : Q$	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

Tabular programs and types are identified up to α-conversion of internal column names and variables bound by **for**-loops.

14.5.2 Type Well-formedness and Expression Types

We begin by discussing the well-formedness rules for environments and column and table types and typing rules for index expressions (which are mutually dependent). We do not present all the rules in detail to save space.

The judgment $\Gamma \vdash \diamond$ holds if the variable names in Γ are unique and all (column and table) types in Γ are well-formed. The column type well-formedness judgment $\Gamma \vdash T$ requires all index expressions in T to be deterministic integers well-formed in Γ. For instance, the well-formedness rule for $T = U[e]$ has the following form:

(Type Array)

$$\frac{\Gamma \vdash U \quad \Gamma \vdash^{\textsf{static}} e : \textsf{int} \;!\; \textsf{det}}{\Gamma \vdash U[e]}$$

https://doi.org/10.1017/9781108770750.015 Published online by Cambridge University Press

The judgment $\Gamma \vdash^{pc} e : T$ states that the index expression e has type T in Γ and only depends on data at level up to pc. Recall that **static** $<$ **inst**. Typing rules for constants and table sizes are trivial, but if e is a variable, then it can correspond to either a variable in Γ or the last column of a Q-type in Γ:

Selected Rules for Index Expressions:

(INDEX VAR) (for $\ell \leq pc$)

$$\frac{\Gamma \vdash \diamond \qquad \Gamma = \Gamma_1, x :^{\ell} T, \Gamma_2}{\Gamma \vdash^{pc} x : T}$$

(FUNREFRET)

$$\frac{\begin{array}{l}\Gamma \vdash \diamond \qquad \Gamma = \Gamma', x : Q, \Gamma'' \\ Q = Q' @ [(\mathsf{ret} \triangleright y : T\ \ell\ \mathbf{output})] \qquad \ell \leq pc\end{array}}{\Gamma \vdash^{pc} x : T}$$

Next, we define well-formedness rules for Q-types and schema types:

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)

$$\frac{\begin{array}{l}\Gamma \vdash T \qquad \Gamma, x :^{\ell} T \vdash Q \\ c \notin \mathsf{names}(Q)\end{array}}{\Gamma \vdash (c \triangleright x : T\ \ell\ \mathbf{input}) :: Q}$$

(TABLE TYPE OUTPUT)

$$\frac{\begin{array}{l}\Gamma \vdash T \qquad \Gamma, x :^{\ell} T \vdash Q \\ c \notin \mathsf{names}(Q)\end{array}}{\Gamma \vdash (c \triangleright x : T\ \ell\ \mathbf{output}) :: Q}$$

(SCHEMA TYPE [])

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash [\,]}$$

(SCHEMA TYPE TABLE)

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

These rules simply require all column types in a Q-type and all table types in a schema type to be well-formed (in the environments formed by preceding columns and tables), all local identifiers to be unique and all field names to be unique within the Q-types in which they are defined. Tables in a schema must also satisfy the **table** predicate.

Every expression in Tabular belongs to one of the three spaces **det**, **rnd** and **qry**, determined by the expression's type. We want to allow information flow from **det** to **rnd** space, because it is harmless to use a deterministic value where a value potentially "tainted" by randomness is expected. Similarly, we want to allow flow from **det** to **qry**. However, since we assume **qry** columns to be deterministic given the inferred posterior distributions of **rnd** columns, we do not allow **qry** columns to reference **rnd** columns directly – information flow from **rnd** to **qry** is only possible via the **infer** operator, which references posterior distributions of random variables, rather than random variables themselves.

We also disallow flows from **qry** to **det** and **rnd**, because we want to ensure that a run of a Tabular program consists of a single round of inference, determining the posterior distributions of **rnd** columns, followed by a single post-processing phase, which computes the values of **qry** columns given the (already known) approximate distributions of **rnd** columns. Tabular does not support nested inference, where, for instance, a **rnd** column used in the second round of inference could depend on a **qry** column computed after the first round of inference.

We embed these restrictions in the type system by means of a subtyping relation on column types. We first define a preorder $\leq$ on spaces as the least reflexive relation satisfying $\textbf{det} \leq \textbf{rnd}$ and $\textbf{det} \leq \textbf{qry}$. We also define a (partial) least upper bound $\mathit{spc} \vee \mathit{spc}'$.

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

$\mathit{spc} \vee \mathit{spc} = \mathit{spc} \qquad \textbf{det} \vee \textbf{rnd} = \textbf{rnd} \qquad \textbf{det} \vee \textbf{qry} = \textbf{qry}$

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

We can lift the $\vee$ operation to types in the straightforward way.

We define the subtyping judgment $\Gamma \vdash T <: U$ to hold if and only if both T and U are well-formed in Γ, they are of the same form and $\mathrm{space}(T) \leq \mathrm{space}(U)$.

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

(Deref Static)

$$\frac{\begin{array}{l}\Gamma \vdash \diamond \qquad \Gamma = \Gamma', t : Q, \Gamma'' \\ Q = Q' @ [(c \triangleright x : T \ \textbf{static}\ \mathit{viz})] @ Q''\end{array}}{\Gamma \vdash^{pc} t.c : T}$$

(Deref Inst)

$$\frac{\begin{array}{l}\Gamma \vdash^{pc} E : \textbf{link}(t) \ ! \ \mathit{spc} \\ \Gamma = \Gamma', t : Q, \Gamma'' \\ Q = Q' @ [(c \triangleright x : T \ \textbf{inst}\ \mathit{viz})] @ Q''\end{array}}{\Gamma \vdash^{pc} E : t.c : T \vee \mathit{spc}}$$

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

$$\frac{\begin{array}{l}D_{\textbf{rnd}} : [x_1 : T_1, \ldots, x_m : T_m](c_1 : U_1, \ldots, c_n : U_n) \rightarrow T \\ \Gamma \vdash^{\textbf{static}} e_i : T_i \quad \forall i \in 1..m \quad \Gamma \vdash^{pc} F_j : \sigma(U_j) \quad \forall j \in 1..n \quad \Gamma \vdash \diamond \\ \{x_1, \ldots, x_m\} \cap (\bigcup_i \mathrm{fv}(e_i)) = \varnothing \quad x_i \neq x_j \text{ for } i \neq j\end{array}}{\Gamma \vdash^{pc} D[e_1, \ldots, e_m](F_1, \ldots, F_n) : \sigma(T)}$$

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

$$\frac{\begin{array}{l}D_{\textbf{qry}} : [x_1 : T_1, \ldots, x_m : T_m](c_1 : U_1, \ldots, c_n : U_n) \rightarrow T \\ \Gamma \vdash^{\textbf{static}} e_i : T_i \quad \forall i \in 1..m \quad \Gamma \vdash^{pc} E : \sigma(T) \quad j \in 1..n \\ \{x_1, \ldots, x_m\} \cap (\bigcup_i \mathrm{fv}(e_i)) = \varnothing \quad x_i \neq x_j \text{ for } i \neq j\end{array}}{\Gamma \vdash^{pc} \textbf{infer}.D[e_1, \ldots, e_m].c_j(E) : \sigma(U_j)}$$

(FunRef)

$$\frac{\begin{array}{l}\Gamma \vdash \diamond \qquad \Gamma = \Gamma', x : Q, \Gamma'' \\ Q = Q' @ [(c \triangleright y : T \ \ell \ \mathit{viz})] @ Q'' \\ \ell \leq pc \quad c \neq \mathsf{ret}\end{array}}{\Gamma \vdash^{pc} x.c : T}$$

Above, we present some of the non-standard typing rules for basic model expressions. Most of them are similar to the typing rules of Fun (Borgström et al., 2013), the language on which the grammar of expressions is based, except that they also handle spaces. We also need to add rules for dereference operators, function column accesses and the **infer** primitive.

The rule (Deref Static) checks that there is an entry for table t in the environment and that its Q-type has column c with type T. (Deref Inst) is similar, except that it typechecks a reference to an **inst**-level column. The index E must be an integer bounded by the size of table t. An instance dereference is only deterministic if both the index and the reference column are deterministic, and a reference to the value of a deterministic column at a random index (or vice versa) is random (and similarly for queries), so we need to join the type of the referenced column with the space of the index.

The (Random) rule requires all hyperparameters $e_1, \ldots, e_m$ of a distribution to be static and have the right types, as specified by the distribution signature. Since the types $U_1, \ldots, U_n$ of parameters and the output type T may depend on these hyperparameters, we need to substitute their values in these types. This is done by the σ operator. The expressions $F_1, \ldots, F_n$ defining parameter values must check against the corresponding types $U_1, \ldots, U_n$ in the signature of the distribution with hyperparameter values substituted by σ. The requirements that the (α-convertible) formal hyperparameter names $x_1, \ldots, x_m$ and free variables in $e_1, \ldots, e_m$ are disjoint and that no hyperparameter name appears twice guarantee that the substitution σ is well-defined.

To see how the (Random) rule works, consider the expression Discrete[2](V) in table Coins in Section 14.2.2. While typechecking the (original, functionless) Coins table, Discrete[2](V) is typeckecked at level **inst** in the environment $\Gamma = \mathsf{V} :^{\mathbf{static}} \mathbf{real}!\mathbf{rnd}[2]$. The signature of the Discrete distribution in space **rnd** is $\mathsf{Discrete}_{\mathbf{rnd}} : [\mathsf{N} : \mathbf{int}!\mathbf{det}](\mathsf{probs} : \mathbf{real}!\mathbf{rnd}[\mathsf{N}]) \to \mathbf{mod}(\mathsf{N})!\mathbf{rnd}$. In Discrete[2](V), the value of the only hyperparameter N is $e_1 = 2$, which obviously checks against the type **int**!**det** in the signature. After substituting this value for N in the type of the only parameter probs, we get **real**!**rnd**[2]. We need to check that $\Gamma \vdash^{\mathbf{inst}} \mathsf{V} : \mathbf{real}!\mathbf{rnd}[2]$. This follows instantly from (Index Var), because V has exactly the type **real**!**rnd**[2] in Γ. Thus, the expression typechecks correctly and the output type, obtained by substituting 2 for N in **mod**(N)!**rnd**, is **mod**(2)!**rnd**.

The (Infer) rule has a similar form to (Random), but instead of typing the distribution arguments, it checks whether the type of the expression E defining the distribution of interest (and normally expected to reference a previous column), matches the output type T in the signature of the distribution D (with hyperparameters $x_1, \ldots, x_m$ again substituted by their values $e_m, \ldots, e_m$). As $\mathbf{infer}.D[e_1, \ldots, e_m].c_j(E)$ is supposed to return the expected value of the parameter c_j of the posterior dis-

https://doi.org/10.1017/9781108770750.015 Published online by Cambridge University Press

tribution of expression E, the type of $\textbf{infer}.D[e_1,\ldots,e_m].c_j(E)$ is the type of the argument c_j in the signature of D, again with the hyperparameter values substituted.

Note that the rule uses the **qry** version of the signature of D, in which the types of arguments are in **qry**-space. This ensures that the type of a post-inference query is in **qry**-space, and thus the query is not part of the probabilistic model.

Let us demonstrate how the rule works by going back to the example in Section 14.2.3. As the table Coins with the additional **qry**-level columns counts and Bias is typechecked, the expression **infer**.Dirichlet[2].pseudocount(V) is typeckeched at level **static** in environment $\Gamma = \mathsf{V} :^{\textbf{static}} \textbf{real}!\textbf{rnd}[2], \mathsf{Flip} :^{\textbf{inst}} \textbf{mod}(2)!\textbf{rnd}$. The signature of Dirichlet in **qry**-space is $\mathsf{Dirichlet}_{\textbf{qry}} : [\mathsf{N} : \textbf{int}!\textbf{det}](\mathsf{pseudocount} : (\textbf{real}!\textbf{qry})[\mathsf{N}]) \to (\textbf{real}!\textbf{rnd})[\mathsf{N}]$. As in the above example for (Random), the value of the only hyperparameter N, which is 2, must be checked at level **static** against the type of N in the signature of $\mathsf{Dirichlet}_{\textbf{qry}}$ – that is, **int**!**det**.

As the posterior distribution of V is expected to be Dirichlet with N = 2, we need to check that the type of the column referenced by V actually matches the output type of Dirichlet with the given hyperparameter, which we obtain by substituting 2 for N in $(\textbf{real}!\textbf{rnd})[\mathsf{N}]$. Looking at the environment Γ, we immediately see that $\Gamma \vdash^{\textbf{static}} \mathsf{V} : \textbf{real}!\textbf{rnd}[2]$ indeed holds.

The parameter of the Dirichlet posterior of V whose expected value the **infer** operator is supposed to return is pseudocount, which has type $(\textbf{real}!\textbf{qry})[\mathsf{N}]$ in the signature. After substituting N by its value, this type becomes $(\textbf{real}!\textbf{qry})[2]$. Hence, this is the type of the entire expression **infer**.Dirichlet[2].pseudocount(V).

The (FunRef) rule defines the type of a column access to be the type of the given column in the type of the reduced table, as long as this column is visible at level pc.

All the other typing rules (including the subsumption rule) are standard.

14.5.3 Model Types

Before we extend the type system to compound models, we define typing rules for function argument lists. The judgment $\Gamma \vdash^{pc} R : Q \to Q'$ means that applying a function of type Q to R at level pc yields a table of type Q'. The typing rules for arguments are presented below. Recall that in functions called at **static** level, the level of every column is reduced to **static**, hence the need to join ℓ with pc in output types.

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

(Arg Input)

$$\frac{\Gamma \vdash^{\ell \wedge pc} e : T \qquad \Gamma \vdash^{pc} R : Q\{^{e}/_{x}\} \to Q'}{\Gamma \vdash^{pc} ((c = e) :: R) : ((c \triangleright x : T\ \ell\ \textbf{input}) :: Q) \to Q'}$$

https://doi.org/10.1017/9781108770750.015 Published online by Cambridge University Press

(Arg Output)

$$\frac{\Gamma, x :^{\ell \wedge pc} T \vdash^{pc} R : Q \to Q' \qquad c \neq \mathsf{ret} \qquad x \notin \mathrm{fv}(R)}{\Gamma \vdash^{pc} R : ((c \triangleright x : T\ \ell\ \mathbf{output}) :: Q) \to ((c \triangleright x : T\ (\ell \wedge pc)\ \mathbf{output}) :: Q')}$$

(Arg Ret)

$$\frac{\Gamma \vdash T}{\Gamma \vdash^{pc} R : (\mathsf{ret} \triangleright x : T\ \ell\ \mathbf{output}) \to (\mathsf{ret} \triangleright x : T\ (\ell \wedge pc)\ \mathbf{output})}$$

The (Arg Input) rule typechecks the argument e, substitutes it for the input variable x and proceeds with checking the rest of R, without copying the input column x to the output type. If the column level ℓ is **static**, e must be a static expression to be a valid argument, and if pc is **static**, then e may be referenced in the subsequent **static** columns of the reduced table, hence we need to typecheck e at level $\ell \wedge pc$. The following rule, (Arg Output), just adds x to the environment (as it may appear in the types of subsequent columns) and proceeds with processing the rest of Q, copying the current column into the output with updated level.

Finally, (Arg Ret) just checks the well-formedness of the type of the output column and updates its level.

In order to simplify typechecking indexed models, we also define an indexing operator for Q-types, which changes the types of all non-input **static rnd** columns in Q into array types.

Indexing a Table Type: $Q[e]$

$\varnothing[e] \triangleq \varnothing$

$((c \triangleright x : T\ \mathbf{inst}\ viz) :: Q)[e] \triangleq (c \triangleright x : T\ \mathbf{inst}\ viz) :: (Q[e]) \quad$ if $x \notin \mathrm{fv}(e)$

$((c \triangleright x : T\ \mathbf{static}\ viz) :: Q)[e] \triangleq (c \triangleright x : T\ \mathbf{static}\ viz) :: (Q[e])$
if $viz = \mathbf{input}$ or $\mathbf{det}(T)$ and $x \notin \mathrm{fv}(e)$

$((c \triangleright x : T\ \mathbf{static}\ viz) :: Q)[e] \triangleq (c \triangleright x : T[e]\ \mathbf{static}\ viz) :: (Q[e])$
if $viz \neq \mathbf{input}$ and $\mathbf{rnd}(T)$ and $x \notin \mathrm{fv}(e)$

We also need to make sure function tables are Core and have no trailing **local** and **input** columns:

Table and Schema Types:

$\mathbf{fun}(\mathbb{T})$ iff $\mathrm{Core}(\mathbb{T})$ and $\mathbb{T} = \mathbb{T}_1 @ [(\mathsf{ret} \triangleright x : T\ \ell\ \mathbf{output}\ E)]$

where @ denotes table concatenation.

The typing rules for (non-simple) models can now be defined as follows:

https://doi.org/10.1017/9781108770750.015 Published online by Cambridge University Press

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

(Model Appl)

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

(Model Indexed)

$$\frac{\Gamma \vdash^{pc} M : Q \quad \Gamma \vdash^{pc} e_{index} : \textsf{mod}(e_{size})\ !\ \textsf{rnd} \quad \textsf{NoQry}(M)}{\Gamma \vdash^{pc} M[e_{index} < e_{size}] : Q[e_{size}]}$$

The (Model Appl) rule typechecks the function table and the argument lists, returning the output type of the argument typing judgment. Meanwhile, (Model Indexed) uses the Q-type indexing to construct the type of an indexed model from the type of its base model. As stated in Section 14.4, only tables with no **qry** columns can be indexed, so we need to ensure that the table nested in M satisfies NoQry.

14.5.4 Table Types

The rules below are used for typechecking both top-level tables and function tables, which can be called from **static** columns, so we need to add the pc level to the typing judgment. To preserve information flow restrictions, a model expression in a column at level ℓ can only reference variables at level at most ℓ. Similarly, expressions in a function at level pc cannot use variables at level greater than pc. Hence, all model expressions are typechecked at level $\ell \wedge pc$.

Tables with Core *columns*

We start with rules for typechecking Core columns. The operator names(Q), used below and in the rest of this section, returns the set of global names of all columns in Q.

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

(Table []) (Table Input)

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash^{pc} [\,] : [\,]} \qquad \frac{\Gamma, x :^{\ell \wedge pc} T \vdash^{pc} \mathbb{T} : Q \quad c \notin \textsf{names}(Q)}{\Gamma \vdash^{pc} (c \triangleright x : T\ \ell\ \textsf{input}\ \epsilon) :: \mathbb{T} : (c \triangleright x : T\ (\ell \wedge pc)\ \textsf{input}) :: Q}$$

(Table Core Output)

$$\frac{\Gamma \vdash^{\ell \wedge pc} E : T \quad \Gamma, x :^{\ell \wedge pc} T \vdash^{pc} \mathbb{T} : Q \quad c \notin \textsf{names}(Q)}{\Gamma \vdash^{pc} (c \triangleright x : T\ \ell\ \textsf{output}\ E) :: \mathbb{T} : (c \triangleright x : T\ (\ell \wedge pc)\ \textsf{output}) :: Q}$$

(Table Core Local) (where $x \notin \text{fv}(Q)$)

$$\frac{\Gamma \vdash^{\ell \wedge pc} E : T \quad \Gamma, x :^{\ell \wedge pc} T \vdash^{pc} \mathbb{T} : Q}{\Gamma \vdash^{pc} (c \triangleright x : T\ \ell\ \textsf{local}\ E) :: \mathbb{T} : Q}$$

https://doi.org/10.1017/9781108770750.015 Published online by Cambridge University Press

The (Table []) rule is obvious. The (Table Input) rule just adds the variable x to the environment (at level $\ell \wedge viz$) and checks the rest of the table. The (Table Core Output) rule checks the model expression E and then typechecks the rest of the table in the environment extended with x. The type of the current column (with level joined with pc) is concatenated with the (recursively derived) type of the rest of the table. (Table Core Local) is similar to (Table Core Output), except that the type of the current column does not appear in the table type and x cannot be free in Q (otherwise Q could contain a variable not defined in the environment Γ in the conclusion of the rule).

Example: checking Core **Tabular functions** To illustrate how the typing rules for Core tables work, recall the functions CDiscrete from Section 14.2.2 and CGaussian from Section 14.4. In this and the following examples, we will use the same column-based notation for Q-types as for Tabular tables.

The function CDiscrete has the following form, with local and field names:

fun CDiscrete			
N ▹ N	**int**!**det**	**static input**	
R ▹ R	**real**!**det**	**static input**	
V ▹ V	**real**!**rnd**[N]	**static output**	Dirichlet[N]([**for** i < N → R])
ret ▹ ret	**mod**(N)!**rnd**	**output**	Discrete[N](V)

To typecheck CDiscrete in an empty environment at level **inst**, we first add the arguments N and R to the environment, by applying (Table Input).

Now, let $\Gamma = \mathsf{N} :^{\mathbf{static}} \mathbf{int}\,!\,\mathbf{det}, \mathsf{R} :^{\mathbf{static}} \mathbf{real}\,!\,\mathbf{det}$. Then, by inspecting the signature of Dirichlet and applying (Random), we can show that

$$\Gamma \vdash^{\mathbf{inst}} \mathsf{Dirichlet[N]([}\mathbf{for}\ \mathsf{i} < \mathsf{N} \to \mathsf{R])} : \mathbf{real}\ !\ \mathbf{rnd}\mathsf{[N]}$$

By applying (Random) again, we get

$$\Gamma, \mathsf{V} :^{\mathbf{static}} \mathbf{real}\ !\ \mathbf{rnd}\mathsf{[N]} \vdash^{\mathbf{inst}} \mathsf{Discrete[N](V)} : \mathbf{mod}\mathsf{(N)}\ !\ \mathbf{rnd}$$

By (Table Core Output), the last column has type

ret ▹ ret	**mod**(N)!**rnd**	**output**

in the environment $\Gamma, \mathsf{V} :^{\mathbf{static}} \mathbf{real}\,!\,\mathbf{rnd}\mathsf{[N]}$. Applying (Table Core Output) again adds the column

V ▹ V	**real**!**rnd**[N]	**static output**

to this type. Finally, by applying (Table Input) twice, we get the type of CDiscrete:

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

Similarly, CG can be shown to have the following type in the empty environment:

M ▹ M	**real**!**det**	**static input**
P ▹ P	**real**!**det**	**static input**
Mean ▹ Mean	**real**!**rnd**	**static output**
Prec ▹ Prec	**real**!**rnd**	**static output**
ret ▹ ret	**real**!**rnd**	**output**

Example: typing function applications Recall the coin flip example from Section 14.2.2, shown here with double column names:

table Coins			
Flip ▹ Flip	**int**!**rnd**	**output**	CDiscrete(N=2, R=1.0)

This example contains a single call to CDiscrete. By the argument typing rules, we have

$$\varnothing \vdash^{\textsf{inst}} (\mathsf{N} = 2, \mathsf{R} = 1.0) : Q_{CD} \to Q'_{CD}$$

where Q_{CD} is the type of CDiscrete, shown above, and Q_{CD}' is the type of the reduced function application, having the following form:

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

By (Model Appl), the type of the function application is Q'_{CD}:

$$\varnothing \vdash^{\textsf{inst}} \mathsf{CDiscrete}(\mathsf{N} = 2, \mathsf{R} = 1.0) : Q'_{CD}$$

Example: indexing model types In the Old Faithful example, we applied indexing [cluster $<$ 2] to the application CG(M $=$ 0.0, P $=$ 1.0). It can be easily shown (like in the example above) that in any environment Γ, this application has the following type Q'_{CG}:

Mean ▹ Mean	**real**!**rnd**	**static output**
Prec ▹ Prec	**real**!**rnd**	**static output**
ret ▹ ret	**real**!**rnd**	**output**

According to the (Model Indexed) rule, in an environment Γ such that $\Gamma \vdash^{\textsf{inst}}$ cluster : **mod** ! **rnd**, the indexed application CG(M $=$ 0.0, P $=$ 1.0)[cluster $<$ 2] has the following type:

Mean ▹ Mean	**real**!**rnd**[2]	**static output**
Prec ▹ Prec	**real**!**rnd**[2]	**static output**
ret ▹ ret	**real**!**rnd**	**output**

Full Tabular Tables

To typecheck columns with non-basic models, we need a prefixing operator for Q-types and two additional rules.

Prefixing function type column names: $c.Q$

$$c.((d \triangleright x : T\ \ell\ viz) :: Q) = (c.d \triangleright x : T\ \ell\ viz) :: c.Q \qquad \text{if } d \neq \mathsf{ret}$$
$$c.([(\mathsf{ret} \triangleright x : T\ \ell\ viz)]) = [(c \triangleright x : T\ \ell\ viz)]$$
$$c.([(d \triangleright x : T\ \ell\ viz)]) = [(c.d \triangleright x : T\ \ell\ viz)] \qquad \text{if } d \neq \mathsf{ret}$$

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

(Table Output)

$$\frac{\Gamma \vdash^{\ell \wedge pc} M : Q_c \quad \Gamma, x : Q_c \vdash^{pc} \mathbb{T} : Q \quad Q_c = Q'_c @ [(\mathsf{ret} \triangleright y : T\ \ell'\ \textbf{output})] \quad \mathsf{names}(c.Q_c) \cap \mathsf{names}(Q) = \varnothing}{\Gamma \vdash^{pc} (c \triangleright x : T\ \ell\ \textbf{output}\ M) :: \mathbb{T} : (c.Q_c)@Q}$$

(Table Local)

$$\frac{\Gamma \vdash^{\ell \wedge pc} M : Q_c \quad \Gamma, x : Q_c \vdash^{pc} \mathbb{T} : Q \quad Q_c = Q'_c @ [(\mathsf{ret} \triangleright y : T\ \ell'\ \textbf{output})]}{\Gamma \vdash^{pc} (\epsilon \triangleright x : T\ \ell\ \textbf{local}\ M) :: \mathbb{T} : Q}$$

The (Table Output) rule typechecks the model M and then recurses into the rest of the table with the environment extended with the type Q_c of M, assigned to x. Note that local attributes of M cannot be referenced in $\mathbb{T}$. This is a design choice – local columns in functions are only meant to be used locally. (Table Local) is similar, except it does not export the type of the model.

Example: typing tables with compound models Recall the coin flip model:

table Coins			
Flip ▹ Flip	**mod**(2)!**rnd**	**output**	CDiscrete(N=2, R=1.0)

We have already shown that the application CDiscrete(N = 2, R = 1.0) has the following type:

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

By (Table Output), the type of the Coins table is:

Flip.V ▹ V	**real**!**rnd**[2]	**static output**
Flip ▹ Flip	**mod**(2)!**rnd**	**output**

Similarly, we can show that the Old Faithful model from the beginning of Section 14.4.1 has the following type:

cluster.V ▹ V	**real**!**rnd**[2]	**static output**
cluster ▹ cluster	**mod**(2)!**rnd**	**output**
duration.Mean ▹ Mean	**real**!**rnd**[2]	**static output**
duration.Prec ▹ Prec	**real**!**rnd**[2]	**static output**
duration ▹ duration	**real**!**rnd**	**output**
time.Mean ▹ Mean	**real**!**rnd**[2]	**static output**
time.Prec ▹ Prec	**real**!**rnd**[2]	**static output**
time ▹ time	**real**!**rnd**	**output**

https://doi.org/10.1017/9781108770750.015 Published online by Cambridge University Press

Example: accessing function fields Let us consider once again the simplified version of the Old Faithful model with an additional column containing a function field access:

table Faithful			
duration ▹ x	**real**!**rnd**	**output**	CG(M=0.0, P=1.0)
time ▹ x’	**real**!**rnd**	**output**	CG(M=60.0, P=1.0)
duration_mean ▹ z	**real**!**qry**	**output**	**infer**.Gaussian.mean(x.Mean)

As shown before, each application of CG has the following type Q'_{CG}:

Mean ▹ Mean	**real**!**rnd**	**static output**
Prec ▹ Prec	**real**!**rnd**	**static output**
ret ▹ ret	**real**!**rnd**	**output**

According to the typing rules, if the initial typing environment is empty, the final column is checked in the environment $\Gamma = x : Q'_{CG}, x' : Q'_{CG}$. This final column must be typechecked by the (Table Core Output) rule, which requires that

$$\Gamma \vdash^{\mathbf{inst}} \mathbf{infer}.\mathsf{Gaussian.mean(x.Mean)} : \mathbf{real} \,!\, \mathbf{rnd}$$

By (Infer), this only holds if

$$\Gamma \vdash^{\mathbf{inst}} \mathsf{x.Mean} : \mathbf{real} \,!\, \mathbf{rnd}$$

The environment Γ can be easily shown to be well-formed. Since x has type Q'_{CG} in the environment, and this Q-type has a column with field name Mean and type **real** ! **rnd**, the above judgment can be derived with (FunRef).

14.5.5 Schema Types

We round off the description of the type system with the following two self-explanatory rules for schemas:

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

(Schema [])

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash [\,] : [\,]}$$

(Schema Table)

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

Top-level tables in a schema are typechecked at level **inst**, because they can define both **static** and **inst**-level columns. The table typing judgment only includes the level parameter because it is also used for typing functions, which can be called from **static** columns.

https://doi.org/10.1017/9781108770750.015 Published online by Cambridge University Press

14.5.6 Type Soundness and Termination of Reduction

In this section, we present the key property of the reduction system: every well-typed schema reduces to a Core schema with the same type. To prove the type soundness property, we need to state and prove three separate propositions: type preservation, progress and termination of reduction. The proofs of these properties are mostly standard inductive proofs and are omitted in this chapter. They can be found in Szymczak (2018).

The type preservation proposition states that if a schema can be reduced, this reduced schema is well-typed and has the same type as the original schema:

Proposition 14.1 (Type preservation)
(1) *If* $\Gamma \vdash^{pc} M : Q$ *and* $M \to M'$, *then* $\Gamma \vdash^{pc} M' : Q$
(2) *If* $\Gamma \vdash^{\mathsf{inst}} \mathbb{T} : Q$ *and* $\mathbb{T} \to \mathbb{T}'$, *then* $\Gamma \vdash^{\mathsf{inst}} \mathbb{T}' : Q$
(3) *If* $\Gamma \vdash \mathbb{S} : Sty$ *and* $\mathbb{S} \to \mathbb{S}'$, *then* $\Gamma \vdash \mathbb{S}' : Sty$.

The progress property states that every well-typed schema which is not in Core form can be reduced.

Proposition 14.2 (Progress)
(1) *If* $\Gamma \vdash^{pc} \mathbb{T} : Q$ *then either* $\mathsf{Core}(\mathbb{T})$ *or there is* $\mathbb{T}'$ *such that* $\mathbb{T} \to \mathbb{T}'$.
(2) *If* $\Gamma \vdash^{pc} \mathbb{S} : Sty$ *then either* $\mathsf{Core}(\mathbb{S})$ *or there is* $\mathbb{S}'$ *such that* $\mathbb{S} \to \mathbb{S}'$.

The final property needed for the type soundness theorem is termination of reduction:

Proposition 14.3 (Termination) *There does not exist an infinite chain of reductions* $\mathbb{S}_1 \to \mathbb{S}_2 \to \cdots$.

By putting these propositions together, we obtain the key theoretical result of this chapter, the type soundness theorem (where we write $\to^*$ for the reflexive and transitive closure of the reduction relation):

Theorem 14.4 *If* $\varnothing \vdash \mathbb{S} : Sty$, *then* $\mathbb{S} \to^* \mathbb{S}'$ *for some unique* $\mathbb{S}'$ *such that* $\mathsf{Core}(\mathbb{S}')$ *and* $\varnothing \vdash \mathbb{S}' : Sty$.

Proof By Propositions 14.1 and 14.2, we can construct a maximal chain of reductions $\mathbb{S} \to \mathbb{S}_1 \to \mathbb{S}_2 \cdots$ such that $\varnothing \vdash \mathbb{S}_i : Sty$ for all i and either $\mathsf{Core}(\mathbb{S}_i)$ or $\mathbb{S}_i \to \mathbb{S}_{i+1}$. By Proposition 14.3, we know that this chain must be finite, so we must have $\mathsf{Core}(\mathbb{S}_i)$ for some $\mathbb{S}_i$. The uniqueness of this $\mathbb{S}_i$ follows from the determinacy of the reduction rules. □

https://doi.org/10.1017/9781108770750.015 Published online by Cambridge University Press

14.6 Conclusions

We have presented a new, significantly extended version of the Tabular schema-based probabilistic programming language, with user-defined functions serving as reusable, modular model components, a primitive for computing quantities depending on inference results, useful in decision theory, and dependent types for catching common modelling errors.

We endowed the language with a rigorous metatheory, strengthening its design. We have defined a system of structural types, in which each table or model type shows the variables used in the model, their domains, determinacies, numbers of instances (one or many) and roles they play in the model. We have shown how to reduce compound models to the Core form, directly corresponding to a factor graph, by providing a set of reduction rules akin to operational semantics in conventional languages, and have proven that this operation is type-sound.

One possible direction of future work is adding support for inference in time-series models. Another possible extension is to allow nested inference Rainforth (2018); Mantadelis and Janssens (2011) by extending the lattice of spaces, so that the distributions computed in one run of inference could be queried by the probabilistic model "active" in the following run. The new lattice would include indexed spaces of the form $\mathbf{rnd}_i$ and $\mathbf{qry}_i$, where the result of **infer** applied to a random variable in $\mathbf{rnd}_i$ would be in $\mathbf{qry}_i$ and data in $\mathbf{rnd}_{i+1}$ could reference columns in space $\mathbf{qry}_i$.

References

Bingham, Eli, Chen, Jonathan P., Jankowiak, Martin, Obermeyer, Fritz, Pradhan, Neeraj, Karaletsos, Theofanis, Singh, Rohit, Szerlip, Paul, Horsfall, Paul, and Goodman, Noah D. 2018. Pyro: Deep Universal Probabilistic Programming. *arXiv preprint arXiv:1810.09538*.

Borgström, Johannes, Gordon, Andrew D., Greenberg, Michael, Margetson, James, and Gael, Jurgen Van. 2013. Measure Transformer Semantics for Bayesian Machine Learning. *Logical Methods in Computer Science*, **9**(3). Preliminary version at ESOP'11.

Borgström, Johannes, Gordon, Andrew D., Ouyang, Long, Russo, Claudio, Ścibior, Adam, and Szymczak, Marcin. 2016. Fabular: Regression Formulas As Probabilistic Programming. Pages 271–283 of: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '16. New York, NY, USA: ACM.

Carpenter, Bob, Gelman, Andrew, Hoffman, Matthew D., Lee, Daniel, Goodrich, Ben, Betancourt, Michael, Brubaker, Marcus, Guo, Jiqiang, Li, Peter, and Riddell, Allen. 2017. Stan: A probabilistic programming language. *Journal of Statistical Software*, **76**(1).

Georgoulas, Anastasis, Hillston, Jane, Milios, Dimitrios, and Sanguinetti, Guido.

2014. Probabilistic Programming Process Algebra. Pages 249–264 of: *Quantitative Evaluation of Systems - 11th International Conference, QEST 2014, Florence, Italy, September 8-10, 2014. Proceedings*.

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

Goodman, Noah, Mansinghka, Vikash K., Roy, Daniel M., Bonawitz, Keith, and Tenenbaum, Joshua B. 2008. Church: a language for generative models. Pages 220–229 of: *Uncertainty in Artificial Intelligence (UAI'08)*. AUAI Press.

Goodman, Noah D., and Stuhlmüller, Andreas. 2014. *The Design and Implementation of Probabilistic Programming Languages*. http://dippl.org.

Gordon, Andrew D., Russo, Claudio, Szymczak, Marcin, Borgström, Johannes, Rolland, Nicolas, Graepel, Thore, and Tarlow, Daniel. 2014a. *Probabilistic Programs as Spreadsheet Queries*. Tech. rept. MSR–TR–2014–135. Microsoft Research.

Gordon, Andrew D., Graepel, Thore, Rolland, Nicolas, Russo, Claudio, Borgstrom, Johannes, and Guiver, John. 2014b. Tabular: A Schema-driven Probabilistic Programming Language. Pages 321–334 of: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. New York, NY, USA: ACM.

Gordon, Andrew D., Russo, Claudio V., Szymczak, Marcin, Borgström, Johannes, Rolland, Nicolas, Graepel, Thore, and Tarlow, Daniel. 2015. Probabilistic Programs as Spreadsheet Queries. Pages 1–25 of: Vitek, Jan (ed), *Programming Languages and Systems (ESOP 2015)*. Lecture Notes in Computer Science, vol. 9032. Springer.

Harper, Robert, and Lillibridge, Mark. 1994. A Type-theoretic Approach to Higher-order Modules with Sharing. Pages 123–137 of: *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '94. New York, NY, USA: ACM.

Herbrich, Ralf, Minka, Tom, and Graepel, Thore. 2007. TrueSkill™: A Bayesian Skill Rating System. Pages 569–576 of: Schölkopf, B., Platt, J. C., and Hoffman, T. (eds), *Advances in Neural Information Processing Systems 19*. MIT Press.

Hutchison, Dylan. 2016. ModelWizard: Toward Interactive Model Construction. *CoRR*, **abs/1604.04639**.

Mansinghka, Vikash K., Selsam, Daniel, and Perov, Yura N. 2014. Venture: a higher-order probabilistic programming platform with programmable inference. *CoRR*, **abs/1404.0099**.

Mansinghka, Vikash K., Tibbetts, Richard, Baxter, Jay, Shafto, Patrick, and Eaves, Baxter. 2015. BayesDB: A probabilistic programming system for querying the probable implications of data. *CoRR*, **abs/1512.05006**.

Mantadelis, Theofrastos, and Janssens, Gerda. 2011. Nesting Probabilistic Inference. *CoRR*, **abs/1112.3785**.

https://doi.org/10.1017/9781108770750.015 Published online by Cambridge University Press

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

Minka, Thomas P. 2001. Expectation Propagation for approximate Bayesian inference. Pages 362–369 of: *Uncertainty in Artificial Intelligence (UAI'01)*. Morgan Kaufmann.

Minka, Tom, and Winn, John. 2009. Gates. Pages 1073–1080 of: Koller, D., Schuurmans, D., Bengio, Y., and Bottou, L. (eds), *Advances in Neural Information Processing Systems 21*. Curran Associates, Inc.

Nori, Aditya V., Hur, Chung-Kil, Rajamani, Sriram K., and Samuel, Selva. 2014. R2: An Efficient MCMC Sampler for Probabilistic Programs. Pages 2476–2482 of: *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*. AAAI'14. AAAI Press.

Pollack, Robert. 2002. Dependently Typed Records in Type Theory. *Formal Aspects of Computing*, **13**, 386–402.

Rainforth, Tom. 2018. Nesting Probabilistic Programs. Pages 249–258 of: Globerson, Amir, and Silva, Ricardo (eds), *Proceedings of the Thirty-Fourth Conference on Uncertainty in Artificial Intelligence, UAI 2018, Monterey, California, USA, August 6-10, 2018*. AUAI Press.

Ścibior, Adam, Ghahramani, Zoubin, and Gordon, Andrew D. 2015. Practical probabilistic programming with monads. Pages 165–176 of: Lippmeier, Ben (ed), *Proceedings of Haskell 2015*. ACM.

Siddharth, N., Paige, Brooks, van de Meent, Jan-Willem, Desmaison, Alban, Goodman, Noah D., Kohli, Pushmeet, Wood, Frank, and Torr, Philip. 2017. Learning Disentangled Representations with Semi-Supervised Deep Generative Models. Pages 5927–5937 of: Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R. (eds), *Advances in Neural Information Processing Systems 30*. Curran Associates, Inc.

Szymczak, Marcin. 2018. *Programming Language Semantics as a Foundation for Bayesian Inference*. Ph.D. thesis, University of Edinburgh.

Vaglica, V., Sajeva, M., McGough, H. N., Hutchison, D., Russo, C., Gordon, A. D., Ramarosandratana, A. V., Stuppy, W., and Smith, M. J. 2017. Monitoring internet trade to inform species conservation actions. *Endangered Species Research*, **32**, 223–235.

Van den Broeck, Guy, Thon, Ingo, van Otterlo, Martijn, and De Raedt, Luc. 2010. DTProbLog: A Decision-theoretic Probabilistic Prolog. Pages 1217–1222 of: *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*. AAAI'10. AAAI Press.

Wood, Frank, Meent, Jan Willem, and Mansinghka, Vikash. 2014. A New Approach to Probabilistic Programming Inference. Pages 1024–1032 of: Kaski, Samuel, and Corander, Jukka (eds), *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics*. Proceedings of Machine Learning Research, vol. 33. Reykjavik, Iceland: PMLR.

https://doi.org/10.1017/9781108770750.015 Published online by Cambridge University Press

Wu, Mike, Perov, Yura N., Wood, Frank D., and Yang, Hongseok. 2016. Spreadsheet Probabilistic Programming. *CoRR*, **abs/1606.04216**. (see also the Scenarios tool at `invrea.com`).

https://doi.org/10.1017/9781108770750.015 Published online by Cambridge University Press