



THE UNIVERSITY *of* EDINBURGH

This thesis has been submitted in fulfilment of the requirements for a postgraduate degree (e.g. PhD, MPhil, DClinPsychol) at the University of Edinburgh. Please note the following terms and conditions of use:

This work is protected by copyright and other intellectual property rights, which are retained by the thesis author, unless otherwise stated.

A copy can be downloaded for personal non-commercial research or study, without prior permission or charge.

This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the author.

The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the author.

When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given.

Programming Language Semantics as a Foundation for Bayesian Inference

Marcin Szymczak

Doctor of Philosophy
Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh
2018

Abstract

Bayesian modelling, in which our prior belief about the distribution on model parameters is updated by observed data, is a popular approach to statistical data analysis. However, writing specific inference algorithms for Bayesian models by hand is time-consuming and requires significant machine learning expertise.

Probabilistic programming promises to make Bayesian modelling easier and more accessible by letting the user express a generative model as a short computer program (with random variables), leaving inference to the generic algorithm provided by the compiler of the given language. However, it is not easy to design a probabilistic programming language correctly and define the meaning of programs expressible in it. Moreover, the inference algorithms used by probabilistic programming systems usually lack formal correctness proofs and bugs have been found in some of them, which limits the confidence one can have in the results they return.

In this work, we apply ideas from the areas of programming language theory and statistics to show that probabilistic programming can be a reliable tool for Bayesian inference. The first part of this dissertation concerns the design, semantics and type system of a new, substantially enhanced version of the Tabular language. Tabular is a schema-based probabilistic language, which means that instead of writing a full program, the user only has to annotate the columns of a schema with expressions generating corresponding values. By adopting this paradigm, Tabular aims to be user-friendly, but this unusual design also makes it harder to define the syntax and semantics correctly and reason about the language. We define the syntax of a version of Tabular extended with user-defined functions and pseudo-deterministic queries, design a dependent type system for this language and endow it with a precise semantics. We also extend Tabular with a concise formula notation for hierarchical linear regressions, define the type system of this extended language and show how to reduce it to pure Tabular.

In the second part of this dissertation, we present the first correctness proof for a Metropolis-Hastings sampling algorithm for a higher-order probabilistic language. We define a measure-theoretic semantics of the language by means of an operationally-defined density function on program traces (sequences of random variables) and a map from traces to program outputs. We then show that the distribution of samples returned by our algorithm (a variant of “Trace MCMC” used by the Church language) matches the program semantics in the limit.

Lay Summary

Bayesian probabilistic modelling, in which the user designs a model expressing how they believe some observable data is generated from some unknown parameters, is one of the most popular approaches to machine learning. However, implementing an efficient inference algorithm, calculating the expected values of unknown parameters, for a given probabilistic model, can be very difficult and time-consuming and require significant knowledge of machine learning and statistics. Meanwhile, there are many professionals who are not machine learning experts but would still like to apply probabilistic modelling to problems in their areas. Probabilistic programming aims to make Bayesian modelling more accessible by letting the user express the desired model as a program in a given probabilistic language—the expected values of unknown parameters are then computed automatically by the inference engine of the language.

This dissertation aims to advance the state of probabilistic programming and consists of two parts. In the first part, we present a substantially extended version of a particular existing probabilistic language, called Tabular, which, instead of extending a general-purpose language with features for probabilistic programming, allows users to specify models as annotated database schemas. We extend this language with user-defined functions, which allow for some reusable model components to be defined just once and used in programs wherever needed. We show how to reduce models with functions to a simpler form on which we can run inference directly. Furthermore we define a dependent type system for Tabular, which catches common modelling errors and helps the user debug a model more quickly. We also endow the language with a semantics, which defines precisely the mathematical meanings of programs, and prove some properties of this extended language. We subsequently extend Tabular with a sub-language which allows expressing hierarchical linear models, a wide and commonly-used class of models, more concisely.

The second part of this dissertation is concerned with correctness of inference algorithms for universal probabilistic languages, which can express a wide class of probabilistic models. These languages typically use inference algorithms such as Metropolis-Hastings Markov chain Monte Carlo, which generates a large number of samples of some unknown quantity to approximate its distribution. Such algorithms usually lack correctness proofs and bugs have been found in some of them. We present the first formal proof of correctness of a variant of Metropolis-Hastings for a functional probabilistic language, which shows that the distribution of samples actually approximates the true distribution of the quantity of interest.

Acknowledgements

First and foremost, I thank my supervisor Andrew D. Gordon, who encouraged me to apply for a PhD programme in the first place and who supported me throughout my studies, even in the most difficult moments. No part of this work could have been completed without his help and guidance.

I also thank my second supervisor David Aspinall for his continuous support and his valuable feedback on my thesis, and Guido Sanguinetti and Sam Staton for agreeing to examine this dissertation and for their useful comments, which helped me improve the final version.

I am grateful to Microsoft Research for funding my scholarship and for offering me two internships.

I thank all the researchers I have been honoured to meet, work with and learn from, in particular (in alphabetical order) Johannes Borgström, Ugo Dal Lago, Aditya Nori, Gordon Plotkin and Claudio Russo.

I am grateful to all my fellow PhD students in the School of Informatics, who made my time as a student so enjoyable.

Finally, I thank my parents Janusz and Małgorzata for always believing in me, and Dorota for keeping my spirits up in the hardest times.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Chapter 4 is based on the paper “Probabilistic Programs as Spreadsheet Queries” published at the 2015 European Symposium on Programming (ESOP), which is joint work with Andrew D. Gordon, Claudio Russo, Johannes Borgström, Nicolas Rolland, Thore Graepel and Daniel Tarlow.

Chapter 5 is based on the paper “Fabular: Regression Formulas as Probabilistic Programming” published at the 2016 Symposium on Principles of Programming Languages (POPL), which is joint work with Johannes Borgström, Andrew D. Gordon, Long Ouyang, Claudio Russo and Adam Ścibior.

Chapters 6 and 7 are based on the paper “A Lambda Calculus Foundation for Universal Probabilistic Programming” published at the 2016 International Conference on Functional Programming (ICFP), which is joint work with Johannes Borgström, Ugo Dal Lago and Andrew D. Gordon.

(Marcin Szymczak)

Table of Contents

1	Introduction	11
1.1	Dissertation Outline	13
1.2	Thesis and Technical Contributions	15
1.3	Publications Included in This Dissertation	17
1.4	Summary of Contributions	18
2	Related Work	21
2.1	Probabilistic Language Design	21
2.1.1	Main Related Languages	22
2.1.2	Spreadsheet and Database-based Systems	24
2.1.3	Brief Summary of Other Systems	25
2.2	Semantics of probabilistic languages	26
2.2.1	Original Research in Probabilistic Languages	26
2.2.2	Current Research on Probabilistic Languages	27
2.3	Correctness of inference in probabilistic programs	29
2.3.1	Multi-site MH for procedural programs [Hur et al., 2015] . . .	30
2.3.2	Single-site MH for abstract programs [Cai, 2016]	31
3	Preliminaries	33
3.1	Probabilistic Inference	33
3.2	Measure Theory	34
3.2.1	Basic Measure Theory	34
3.2.2	A Measure Space on Program Traces	37
3.2.3	Metric and Topological Spaces	41
3.2.4	Subprobability and Probability Kernels	43
3.3	Metropolis-Hastings Sampling in General State Spaces	43
3.3.1	Markov Chains	44

3.3.2	Metropolis-Hastings Markov chain Monte Carlo (MH-MCMC)	45
4	Tabular: A Schema-based Probabilistic Language	47
4.1	Introduction and Examples	48
4.1.1	Probabilistic Programming in Tabular	48
4.1.2	User-Defined, Dependently-Typed Functions	51
4.1.3	Query Variables	53
4.1.4	Critique of the Preliminary Version	53
4.1.5	Contributions	55
4.1.6	Interface and Implementation	56
4.2	Syntax of Tabular	57
4.2.1	Syntax of Databases	57
4.2.2	Syntax of Core Schemas	58
4.2.3	Syntax of Schemas with Functions and Indexing	62
4.3	Reduction to Core Tabular	64
4.3.1	Reducing Function Applications	65
4.3.2	Reducing Indexed Models	72
4.3.3	Reducing Schemas	76
4.4	Type System	77
4.4.1	Syntax of Tabular Types	78
4.4.2	Type Well-formedness and Expression Types	80
4.4.3	Model Types	84
4.4.4	Table Types	86
4.4.5	Schema Types	91
4.4.6	Type Soundness and Termination of Reduction	91
4.5	Semantics	93
4.5.1	Evaluation Environments and Databases	94
4.5.2	Input Database Conformance	94
4.5.3	Semantics of Probabilistic Models	97
4.5.4	The Probability Measures on Random Expressions	106
4.5.5	Semantics of Queries	111
4.5.6	Output Database Conformance	117
4.6	Conclusions	122

5	Fabular: Tabular with Regression Formulae	123
5.1	Linear Regression Formulae in R and Their Limitations	124
5.2	Syntax of the Regression Calculus	125
5.3	Typing Regression Formulae	131
5.4	Fabular = Tabular + Regression Formulae	135
5.4.1	Syntax and Type System of Fabular	136
5.4.2	Translation to Core Tabular	137
5.4.3	Examples	141
5.4.4	Type Soundness for Fabular	143
5.5	Conclusions	146
6	Semantics of a Lambda Calculus with Continuous Distributions	147
6.1	A Probabilistic λ -calculus	149
6.1.1	Big-step Sampling-based Semantics	152
6.1.2	Encoding Church in the Core Calculus	154
6.1.3	Example: Geometric Distribution	156
6.1.4	score and Soft Conditioning	158
6.2	Small-step Semantics	160
6.2.1	Equivalence of Small-step and Big-step Semantics	169
6.3	A Distribution on Program Outcomes	172
6.3.1	Distributions on Random Traces and Program Outcomes	173
6.3.2	Digression: Motivation for Bounded Scores	178
6.4	Conclusions	179
7	Correct Metropolis-Hastings for Functional Probabilistic Programs	181
7.1	A Metropolis-Hastings Sampling Algorithm	182
7.2	Transition Kernel	186
7.3	Correctness of Inference	188
7.3.1	Additional properties of reduction	189
7.3.2	π -Irreducibility	193
7.3.3	Aperiodicity	194
7.3.4	Harris recurrence	194
7.4	Examples	199
7.4.1	Geometric Distribution	199
7.4.2	Linear Regression with <code>flip</code>	200
7.4.3	Linear Regression with <code>score</code>	202

7.5	Discussion of the Algorithm	203
7.5.1	Motivation for Using Multi-Site Inference	204
7.5.2	Problem With Identifying Random Variables	204
7.6	Conclusions	206
8	Conclusions	209
A	Alpha-equivalence of Tabular programs	213
B	Proofs of the Propositions from Section 4.4.6	217
B.1	Proposition 1	217
B.2	Proposition 2	242
B.3	Proposition 3	245
C	Proof of Tabular Output Database Conformance	249
C.1	Random Semantics	249
C.2	Preservation Result for the Query Semantics	258
C.3	Progress Result for Query Semantics	269
D	Proof of Lemma 18 in Chapter 5	277
E	Proofs of Lemmas in Chapters 6 and 7	283
E.1	Deterministic reduction as a measurable function	290
E.2	Small- step reduction as a measurable function	293
E.3	Measurability of P and O	295
E.4	Measurability of <code>peval</code>	303
E.5	Compositionality of <code>peval</code>	307
E.6	Measurability of q and Q	309
	References	317

Chapter 1

Introduction

Over the past half-century, machine learning has turned from a science-fiction fantasy to a ubiquitous technology which we start taking for granted. These days, no-one is astonished by computers using historical data to recognize handwritten postcodes, predict stock markets, understand our speech and even drive our cars. The name “machine learning” is in fact an umbrella term encompassing all sorts of techniques for inferring unknown quantities from existing data. One of the most successful and popular paradigms is *model-based Bayesian machine learning*.

In this paradigm, we need to define the problem by specifying first the *prior* distribution on the unknown parameters (according to which the parameters would be distributed in the lack of any observations) and then a *generative model*, explaining how we believe the observed output values were generated from the parameters. We can then express the probability distribution on the unknown parameters in terms of the prior distribution on the parameters and the output likelihood by using the Bayes rule. But how do we represent such a model? And how can we efficiently perform inference in a non-trivial model?

Implementing a custom inference algorithm for any real problem is not only cumbersome and time-consuming, but also requires a great deal of machine learning expertise. Yet many of the people who need to use probabilistic inference are not machine learning PhDs, but “domain experts” simply wanting to solve some problems in their domains.

Probabilistic programming has been hailed as a way of bringing Bayesian inference to the masses and making machine learning more accessible to domain experts [Gilks et al., 1994, Goodman, 2013, Gordon, 2013]. It enables users to perform probabilistic inference simply by providing a generative description of their model as a computer

program, without having to worry about the underlying inference engine. This way, even non-experts can use probabilistic inference to reason about systems with random behaviours, and experts can do so more rapidly. Because of that, the advent of probabilistic programming has been compared to the revolution in software engineering brought about by compilers and high-level languages [Duvenaud and Lloyd, 2013].

The reality, however, is much different from this idealised picture. Over twenty years after BUGS [Gilks et al., 1994], the first mainstream probabilistic language for Bayesian inference, was developed, the field still seems to be in its infancy, with experts still refusing to abandon their trusted problem-specific inference algorithms and some non-experts unable apply this technology to problems in their areas.

For one thing, most currently available systems are not perfectly suited for people who are not professional programmers. While tools such as Church [Goodman et al., 2008], a higher-order Turing-complete probabilistic language, or BLOG [Milch et al., 2005], a package based on higher-order logic, allow for many complex probabilistic models to be expressed concisely, some business analysts and applied statisticians might find them baffling, because using them requires familiarity with non-standard programming paradigms (such as functional or logic programming). The recently-proposed Tabular language [Gordon et al., 2014], in which models are expressed as annotated database schemas and which can be embedded in spreadsheet applications, represents an attempt to make probabilistic programming more accessible. In the first part of this dissertation, we extend this language with user-defined functions, pseudo-deterministic queries on inference results and an embedded calculus for representing hierarchical linear regressions. We present the syntax of the resulting language, define a reduction relations reducing compound models to the so-called Core form (corresponding directly to factor graphs) and endow the language with a dependent type system, catching common modelling errors. We also define the semantics of Tabular and prove some theoretical results showing that the language is well-behaved.

Moreover, there is little confidence in the results returned by the underlying inference algorithms for probabilistic languages. The claims of validity of many of these inference methods, such as “Trace MCMC” used by Church, are backed by limited case studies rather than formal correctness proofs. To address this issue, we present the first proof of correctness of the Metropolis-Hastings algorithm for a higher-order functional probabilistic programming language. This proof involves defining a semantics of a probabilistic lambda-calculus with recursion, which defines a distribution on output values of the given program.

There are also other problems with implementing probabilistic programming systems, of which the poor performance of inference engines (speed of convergence of MCMC, quality of approximations used by message-passing algorithms, etc.) is the most prominent. However, the speed of convergence of approximate inference algorithms for probabilistic languages has already received a lot of attention in the machine learning community [Paige and Wood, 2014, Yang et al., 2013, Wood et al., 2014, Ritchie et al., 2016, Le et al., 2017]. This dissertation is, instead, focused on the aspects of probabilistic programming which are often overlooked in the quest of speed and performance.

1.1 Dissertation Outline

This section summarises the content of this dissertation.

Chapter 2 summarizes the previous work on probabilistic language design and semantics and describes briefly some of the more popular probabilistic programming systems.

Chapter 3 provides some background knowledge necessary to understand the details of this work, to make the material accessible to people coming from different backgrounds (programming languages, machine learning, statistics).

Chapter 4 presents a new, extended version of Tabular, an easy-to-use schema-driven language. We define the syntax of the language, which includes functions (defined as tables), indexed models and a query operator for post-processing inference results, and give it a basic dependent type system with information flow tracking, to help catch bugs before inference. We provide a reduction relation which reduces programs with compound model expressions into simpler programs in what we call a “Core” form, and we show that this reduction always yields a well-typed program. We define a semantics of Core Tabular in two steps: first, a measure-theoretic semantics computing the joint distribution of all random expressions of interest in the program, and then an operational semantics calculating the approximate values of queries. We show that the output schema containing the computed values of queries is always well-formed. The main technical results are **Theorem 1**, which states that reduction of Tabular models into Core form is type-sound, and **Theorem 2**, stating that the semantics of queries produces valid output databases, conforming to the database schema. The material presented in this chapter is a significant reworking of the paper on which the chapter is based [Gordon et al., 2015], with updated language syntax (to facilitate reasoning

about substitution), a completely new semantics with proper support for conditioning on values of continuous random draws (instead of relying on discretisation of real numbers), modified reduction relation for compound Tabular models and a more rigorous proof of Theorem 1, which required significant effort due to the unusual nature of the Tabular language.

Chapter 5 describes a further extension of Tabular: an embedded calculus for representing hierarchical, generalized linear models, extending the popular formula notation used by several statistical inference packages in the R language. We present a type system for the calculus and embed the calculus in Tabular, which gives a new language called Fabular. We define a translation from Fabular to Core Tabular and prove **Theorem 3**, which states that the type soundness result for Tabular schema reduction extends to Fabular. We also demonstrate the expressiveness and conciseness of Fabular by several examples.

Chapter 6 defines the semantics of a probabilistic lambda-calculus with continuous random variables and conditioning. We first specify an operational semantics reducing an expression to a value deterministically, given a sequence of random draws. We give both big-step and small-step semantics, which are equivalent by **Theorem 4**. We then define a distribution on program traces as an integral of the evaluation function with respect to the stock measure on traces, from which we can get a distribution on program outcomes by a simple measure transformation. We prove **Theorem 5**, stating that the distribution of outcomes of a valid program is a subprobability measure on the space of values in our λ -calculus. The proof is fully rigorous and includes showing the measurability of the functions defined in terms of the semantics—an issue often neglected in similar developments. We also translate a subset of Church, a popular real-life probabilistic functional language, to the core calculus, thus endowing it with a rigorous semantics.

Chapter 7 presents a formal proof of correctness of a variant of the Trace MCMC algorithm for inference in higher-order functional probabilistic programs, showing that the distribution of samples returned by it matches the semantics of the program (as defined in the previous chapter). We define a variant on the Trace MCMC algorithm and use standard results from theoretical statistics to prove **Theorem 6**, which says that the distribution of samples generated by this algorithm converges to the program semantics. Again, the proof includes showing that all Lebesgue-integrated functions are measurable and that all assumptions of the theorems used are satisfied.

1.2 Thesis and Technical Contributions

The central claim of this dissertation is that **probabilistic programming can be a convenient and trustworthy tool for Bayesian inference**. The dissertation includes the following key technical developments:

- (1) A reduction relation reducing Tabular schemas with functions and indexing to a Core form;
- (2) A measure-theoretic semantics of Tabular, defining marginal distributions on queried expressions, and an operational query semantics of Tabular, defining the expected answers to pseudo-deterministic queries;
- (3) A structural dependent type system for Tabular with information flow tracking;
- (4) An extension of Tabular with an embedded hierarchical linear regression calculus and an adaptation of the Tabular type system and reduction relation to the extended language, called Fabular;
- (5) An operational sampling-based semantics of an untyped probabilistic lambda-calculus with continuous distributions and conditioning, which gives rise to a distribution on program outcomes;
- (6) A rigorous proof of convergence of a variant of the Metropolis-Hastings algorithm for a probabilistic lambda-calculus.

These developments lead to research contributions to probabilistic programming, which support the thesis of this dissertation.

Regarding the first three points, we believe the new version of Tabular is the only spreadsheet-based probabilistic language with user-defined functions, a type system and a formal semantics. Other existing systems for probabilistic computation in spreadsheets, such as @Risk (<http://www.palisade.com/risk/>) and Scenarios (<http://www.invrea.com>), lack not only semantics and type systems, but also any scientific publications. The query semantics of Tabular also demonstrates a new way of defining the meaning of probabilistic programs, by focusing not on the distributions, but on actual numerical values that the user may want to extract from the posterior distributions.

Furthermore, regarding the third development, there currently exists no other probabilistic language with statically-checked dependent types. The Stan language features dependent types, but the type constraints are only checked at runtime.

Regarding point 4, the embedding of a linear regression calculus inside a general-purpose probabilistic language is a novel idea. Moreover, this embedding and the reduction to Core Tabular provides the calculus with a rigorously defined semantics, which is also a novel contribution—R’s widely-used formula language lacks any formal semantics, and the meaning of expressions is only explained in words.

As for point 5, defining the distributional semantics of the untyped probabilistic lambda-calculus with continuous distributions and conditioning is an open research problem. The most recent advance in this area is the work by Heunen et al. [2017], who present a denotational semantics of a simply-typed lambda calculus with continuous and discrete observations and conditioning—however, their language, being simply-typed, does not support higher-order recursion. To our best knowledge, our semantics is the first to define distributions on output values for such a language (although, admittedly, it does not define distributions on mathematical functions, treating lambda-abstractions purely syntactically).

Finally, regarding point 6, this dissertation presents, to our best knowledge, the first proof of correctness of a variant of Metropolis-Hastings for a higher-order functional language. Hur et al. [2015] present a proof of correctness of Metropolis-Hastings, but their proof only applies to an imperative, procedural language and is less rigorous than this work—it only shows the reversibility of the constructed Markov chain, disregarding aperiodicity and ϕ -irreducibility (which are proven in this dissertation), and assumes without proof measurability of all functions used. A more general proof, applicable to a functional language, was presented by Cai [2016], but this proof does not deal with language semantics, treating programs as dependent sequences of probability kernels, and this yet unpublished work was made known to me and my collaborators after the paper on which Chapters 6 and 7 are based was accepted for publication.

The main theoretical results of this dissertation are the following:

- **Theorem 1:** reduction of Tabular models with functions and indexing to Core form is type-sound;
- **Theorem 2:** the query semantics of Tabular maps well-typed Tabular schemas and conformant input databases to well-defined output databases;
- **Theorem 3:** Reduction of Fabular models to Core Tabular is type-sound.
- **Theorem 4:** the small-step and big-step semantics of the probabilistic lambda-calculus are equivalent;

- **Theorem 5:** the semantics of the probabilistic lambda-calculus defines a sub-probability measure on output values;
- **Theorem 6:** The distribution of values generated by the variant of Metropolis-Hastings for the probabilistic lambda-calculus presented in this dissertation converges to the semantics of the given program.

1.3 Publications Included in This Dissertation

The technical material included in this dissertation consists mostly of significantly extended and revised content of three published conference papers:

- The starting point for Chapter 4 was the paper “Probabilistic Programs as Spreadsheet Queries” [Gordon et al., 2015] published at the 2015 European Symposium on Programming (ESOP), which was joint work with Andrew D. Gordon, Claudio Russo, Johannes Borgström, Nicolas Rolland, Thore Graepel and Daniel Tarlow. The syntax and type system of Tabular, as well as the reduction rules reducing function applications and indexed models, have been substantially reworked to fix a problem with α -conversion found in the paper and to make the presentation cleaner. The proof of type soundness of reduction to Core Tabular has been updated and made more rigorous. This chapter also features a completely new semantics of Core Tabular, which replaces the slightly inelegant semantics presented in the paper, relying on discretisation of real numbers and an unspecified abstract inference algorithm.
- Chapter 5 is based on the paper “Fabular: Regression Formulas as Probabilistic Programming” [Borgström et al., 2016] published at the 2016 Symposium on Principles of Programming Languages (POPL), which paper was joint work with Johannes Borgström, Andrew D. Gordon, Long Ouyang, Claudio Russo and Adam Ścibior. However, the syntax of the regression calculus presented here has been modified, to make a clear distinction between local variables and globally-visible parameters. The translation of Fabular to Core Tabular has been updated. The proof of Theorem 3, which shows that the translation of Fabular to Tabular is type-sound, is a new contribution and was not included in the aforementioned publication.

- Chapters 6 and 7 are based on the paper “A Lambda Calculus Foundation for Universal Probabilistic Programming” [Borgström et al., 2016] published at the 2016 International Conference on Functional Programming (ICFP), which was joint work with Johannes Borgström, Ugo Dal Lago and Andrew D. Gordon. This chapter uses a different definition of the density of transition kernel of the presented variant of Metropolis-Hastings, which actually corresponds to an implementable sampling procedure, fixes a bug in the examples of inference and presents a more rigorous proof of the ϕ -irreducibility of the transition kernel of the algorithm.

All the publications on which this dissertation is based were written in collaboration with other researchers. However, my contributions were important and are summarised at the end of each chapter. Only the parts of the papers to which I have made significant contributions are included. All the new extensions and improvements upon the work described in the papers are entirely my own work.

1.4 Summary of Contributions

In summary, this dissertation makes the following novel research contributions:

- The design, syntax and semantics of Tabular, the first schema-based probabilistic language supporting user-defined functions;
- The first static dependent type system for a probabilistic programming language, which allows catching modelling errors and includes information flow tracking;
- The first embedding of a hierarchical linear regression calculus in a general-purpose probabilistic language (implicitly endowing the calculus with a semantics);
- A semantics of a higher-order untyped probabilistic lambda-calculus with continuous random draws and soft and hard conditioning, defining distribution on syntactic values;
- The first proof of correctness of a variant of the Metropolis-Hastings inference algorithm for a functional probabilistic language.

These contributions have direct practical significance in times when machine learning is becoming ubiquitous and probabilistic programming is gaining ground as a way of performing probabilistic inference. Reducible user-defined functions enhance the modelling power of Tabular without making inference more complicated. The dependent type system of Tabular guides model creation and speeds up modelling by catching common errors statically, before running inference. The embedding of the regression calculus in Tabular shows how probabilistic languages can be composed, with the resulting language retaining a consistent semantics and type system. The semantics of a higher-order functional language with recursion helps understand the mathematical meaning of arbitrary probabilistic models, including nonparametric models. Finally, proving correctness of inference algorithms is important when probabilistic programming starts being used in safety-critical settings, such as controlling autonomous vehicles.

While this dissertation is theoretical in nature, and the results are mostly theorems, it should be stressed that a significant amount of more practical work has been involved in the research that lead to it, including an early implementation of a typechecker for Tabular, an implementation of Fabular, a preliminary implementation of the variant of Metropolis-Hastings used in Chapter 7, and a compiler translating Tabular programs to Stan. The work on the Fabular implementation was the most substantial part, and in addition to a compiler translating formulas to Core Tabular, it included automatic generation of plots showing quantities of potential interest to the user. This work has not been published nor released and is not presented in this dissertation, but may be included in future releases of Tabular or other derived systems.

Chapter 2

Related Work

In this chapter, we present an overview of the literature on probabilistic programming, in order to show historical advances and the current state-of-the-art in the areas involved and highlight the originality of the material included in this dissertation. We begin by an overview of existing probabilistic programming languages, which aims to show what the Tabular language presented in Chapter 4 brings to the table in terms of expressiveness and ease-of-use, and introduces higher-order functional probabilistic languages, which are the motivation for the work presented in Chapters 6 and 7. We then present the related work on the semantics of probabilistic languages, in order to compare our semantics presented in Chapter 6 to other related developments. Finally, we discuss the scarce existing literature on the correctness of inference in probabilistic languages, to put the developments presented in Chapter 7 in context.

2.1 Probabilistic Language Design

Over the past two decades, many probabilistic programming systems based on different paradigms and using various inference algorithms have seen the light of day. Some of these languages, such as BUGS, are designed to be efficient and easy to use, but only support limited classes of models, while others, including Church, are more flexible and allow defining all computable distributions, but are more complicated and require significant programming expertise.

2.1.1 Main Related Languages

We present here some of the widely used probabilistic languages which are most relevant to this dissertation: the BUGS language, which was the first widely-used probabilistic language, Stan, which, like Tabular, aims to be a feature-rich language accessible to domain experts, and Church, the first well-known functional higher-order probabilistic language which inspired the last two technical chapters of this dissertation.

- **BUGS** (Bayesian inference using Gibbs sampling) [Gilks et al., 1994] was the first mainstream software package providing a generic inference engine for arbitrary, user-defined graphical models, specified in a high-level declarative language. Models in this language are specified by a custom input format and interpreted as factor graphs. As its name implies, BUGS uses Gibbs sampling [Geman and Geman, 1984] for inference. At each step, the value of one parameter is updated, using its full conditional distribution.

BUGS gained some appeal in academia and was applied to statistical problems in many disciplines [Lunn et al., 2009] throughout its lifespan. The language had multiple implementations, most prominent of which were WinBUGS, designed in the mid-1990s, and the later cross-platform implementation called OpenBUGS.

Some application-specific interfaces for BUGS have also been created, such as PKBugs for pharmacokinetic modelling and GeoBugs for spatial modelling.

- **Stan** [Stan Development Team, 2014] is a probabilistic programming package inspired by BUGS which is currently one of the most popular and most actively developed probabilistic programming languages.

Stan programs are compiled to machine code via C++. Unlike BUGS, which updates only one parameter at a time by using Gibbs sampling, Stan can move in any direction in the parameter space in a single step, by using a new algorithm called No-U-Turn-Sampler (NUTS) [Hoffman and Gelman, 2013], a variant of Hamiltonian Monte Carlo. The current version of Stan supports user-defined functions and while-loops, making the language more expressive than BUGS. Functions cannot contain local variable declarations, but recent work by [Gorinova, 2017] extends the language to lift this restriction.

Stan has been highly regarded for its flexibility and its efficient sampling algorithm [Monnahan et al., 2017]. However, one important limitation of this system is that the inference algorithm can only sample continuous variables, which eliminates the possibility of using discrete parameters directly in models. To enhance the expressiveness of the modelling language, Stan supports direct manipulation of the likelihood of the model by the `target += ...` construct.

- **Church** [Goodman et al., 2008] is a Turing-complete higher-order functional probabilistic programming language which supports recursion, discrete and continuous random variables and conditioning. The syntax of the language is based on Scheme [Sperber et al., 2010], a minimalist dialect of Lisp.

The top-level construct in Church is a probabilistic query, consisting of a sequence of function and variable definitions, an output expression to be evaluated and a boolean-valued expression denoting the condition, depending on the sampled random variables, which has to be satisfied for the given program run to be valid.

An important feature of Church is memoisation, which, unlike memoisation in deterministic languages, is a semantically significant construct in Church. When a memoised procedure is first called with given parameters, its return value is remembered, and on each subsequent call to this procedure with the same arguments this stored value is returned, without re-evaluating the function (and resampling random parameters). Stochastic memoisation can be used, for instance, to implement the Dirichlet process, which allows clustering data when the number of clusters is not known in advance.

The original implementation of the language is based on a variant of the Metropolis-Hastings MCMC [Metropolis et al., 1953, Hastings, 1970] algorithm, which in each step, given some trace of a probabilistic program, perturbs the value of one elementary random choice, updates references to it in subsequent probability calculations and refreshes the trace by performing new evaluations if the change has affected the control flow of the program. Wingate et al. [2011] describe the Metropolis-Hastings inference algorithm for Church in more detail and propose a more efficient, “lightweight” implementation, based on a static source-to-source translation turning a Church program into a MCMC inference procedure.

Yang et al. [2014] use tracing and slicing to improve the efficiency of Trace

MCMC in Church. Their optimisations reduce the overhead resulting from having to compute the address of every random variable on each random primitive call and recompute the acceptance ratio at each step of the algorithm.

As an alternative to sampling-based methods, Stuhlmüller and Goodman [2012] also presents an exact, deterministic inference engine for Church, based on dynamic programming.

Since its inception, Church has forked into several derived languages. One of them is **Venture** [Mansinghka et al., 2014], with modified syntax and programmable, compositional inference. Another Church-based language is **Anglican** [Wood et al., 2014], whose original inference engine is based on the particle MCMC algorithm [Andrieu et al., 2010], which, as Wood et al. [2014] show, is more efficient than standard Metropolis-Hastings on some problems. A very promising new approach to inference in Anglican (and probabilistic programs in general), called inference compilation, was presented by Le et al. [2017], who propose using deep neural networks to optimise the proposal distributions from which random variables are sampled.

2.1.2 Spreadsheet and Database-based Systems

In this section, we present two packages for statistical inference which, like Tabular, have spreadsheet-based interfaces.

- **@Risk** (<http://www.palisade.com/risk/>) is a commercial Excel plug-in for probabilistic computation, designed specifically for risk analysis in business. It allows generating large numbers of samples from probabilistic models via the Monte Carlo method, features extensive visualisation tools and supports fitting basic distributions to data, but, crucially, does not seem to support inference in probabilistic models.
- **Scenarios** (<http://www.invrea.com/>) is another Excel plug-in, also aimed at business users, which allows defining probabilistic models within spreadsheets. Unlike @Risk, it allows conditioning on output values and computation of posterior distributions on unknown parameters. However, it has neither formal semantics nor a type system.

2.1.3 Brief Summary of Other Systems

To complete this section, we present here a quick overview of other selected probabilistic programming languages.

- **IBAL** [Pfeffer, 2001]: a Turing-complete language implemented in OCaml, allowing recursive function definitions. It uses variable elimination with memoisation for inference.
- **BLOG** [Milch et al., 2005]: a logic language, designed to reason with an unknown number of objects, allowing nonparametric models. Multiple inference backends, including a Metropolis-Hastings-based one, are available.
- **Autobayes** [Schumann et al., 2008]: a package which generates C++ inference code from Bayesian networks. It uses analytical methods (k-means clustering, Expectation Maximisation and symbolic differentiation) whenever possible, otherwise resorts to numerical optimization.
- **Blaise** [Bonawitz, 2008]: a package using a graphical modelling language. Blaise programs are represented as generalised factor graphs (specifying dependencies between variables), with the plate notation normally used for creating copies of variables replaced with a composition structure, allowing for the number of copies to be used as a variable. Blaise has a customisable, sampling based inference engine.
- **FACTORIE** [McCallum et al., 2009]: a Scala-based language for representing undirected graphical models, requiring users to create factor graphs manually. Uses a variant of MCMC for inference.
- **Figaro** [Pfeffer, 2009]: an object-oriented probabilistic language, also based on Scala. It supports a broad class of models, including ones with unknown number of objects. It is equipped with multiple backends, including Metropolis—Hastings (with custom proposals), exact inference and Expectation Maximization.
- **HANSEI** [Kiselyov and Shan, 2009]: a probabilistic language embedded in OCaml, supporting inference in nonparametric models. HANSEI’s inference algorithm is based on importance sampling.
- **ProbLog** [Broeck et al., 2010]: a probabilistic extension of Prolog.

- **Filzbach** [Purves and Lyutsarev, 2012]: a MCMC-based tool for statistical inference, used mostly in biological models. Highly efficient, but requires a very low-level specification of the model.
- **Fun** [Borgström et al., 2013]: a first-order functional language based on a subset of F#. Fun supports inference in factor graphs, using Infer.NET [Winn and Minka, 2009] as a backend and has been extended with recursion by [Georgoulas et al., 2013] to support inference in continuous-time Markov chains.
- **BayesDB** [Mansinghka et al., 2015]: a probabilistic language based on databases, with a SQL-like interface.
- **ProPPA** [Georgoulas et al., 2014]: a probabilistic language based on the BioPEPA process algebra

2.2 Semantics of probabilistic languages

In this section, we present the related work on the semantics of probabilistic languages, to put our semantics of a probabilistic lambda-calculus presented in Chapter 6 in context. We conclude that no previous work defines distributions on the output values of arbitrary programs in a probabilistic lambda-calculus with higher-order recursion, continuous distributions and soft conditioning, which the semantics presented in this dissertation does.

The literature on the semantics of probabilistic programs is split into two distinct phases. Until the early 90’s, before the appearance of the first widely-used languages for probabilistic inference, most of the research on probabilistic languages was carried out with applications such as randomised algorithms, rather than machine learning, in mind. Hence, most foundational calculi studied then had only discrete distributions and no primitives for conditioning. After the turn of the century, probabilistic programming for machine learning became more widespread, and became the main focus of probabilistic language semantics research.

2.2.1 Original Research in Probabilistic Languages

The pioneering work on semantics of probabilistic programs was a paper by Saheb-Djahromi [1978], which defines the semantics of a higher-order typed functional language with discrete random draws, based on the discrete probabilistic domain. The

author presents a small-step operational semantics of the language, in which small-step reductions are parametrised by reduction probabilities. This semantics induces a distribution on program outcomes. A denotational semantics is also presented and shown equivalent to the distribution defined by the operational semantics.

Kozen [1981] presents two different semantics for a while-language with random numbers. In the first semantics, a program is defined as a (deterministic) partial measurable function taking an initial valuation of its variables concatenated with an infinite vector of random choices, and returning an updated valuation of its variables, together with unused “tail” of the random vector. As it is straightforward to define a measure on infinite sequences, this program can be seen as a measure transformer. The second semantics takes an initial measure on inputs, and computes a measure of outputs directly, in a compositional way. The two semantics are shown equivalent.

Jones and Plotkin [Jones, 1989, Jones and Plotkin, 1989] define the semantics of (abstract) probabilistic computations in terms of continuous evaluations, a generalization of measures. More precisely, they define the spaces of results as inductive partial orders (ipos), which, with continuous functions between them as morphisms, form a category **Ipo**, and an endofunctor \mathcal{V} mapping objects in **Ipo** to evaluations and morphisms to evaluation transformers, which yields a powerdomain of evaluations. They then apply this theory to define the denotational semantics of a simple while-language with discrete probabilistic choice and a functional language with recursion, also with just discrete random draws.

2.2.2 Current Research on Probabilistic Languages

Ramsey and Pfeffer [2002] apply Giry’s probability monad [Giry, 1982] to define denotational semantics of a stochastic lambda-calculus with discrete distributions (easily extensible to continuous ones). Their language supports neither recursion nor conditioning.

Danos and Harmer [2002] define a semantics of probabilistic PCF based on game theory and show the full abstraction result for it.

Park et al. [2005] present an operational semantics of a typed functional language with recursion and higher-order functions, in which distributions are represented in terms of sampling functions. The semantics is parametrized by an infinite “tape” of random numbers on the unit interval, and the reduction relation is defined on tuples of expressions and tapes: intuitively, it takes an expression and a tape, reduces this

expression by taking an element from the tape when a random number is needed, and returns the reduced expression together with the remainder of the tape. The authors show by example how this semantics could be used to define the distribution induced by the program, but stop short of formalizing this idea.

Dal Lago and Zorzi [2012] study operational semantics for the probabilistic λ -calculus with discrete random choice. They define, in inductive and coinductive way, big-step and small-step semantics for both call-by-name and call-by-value λ -calculus, and shows that in each case the big-step and small-step versions coincide.

Cousot and Monerau [2012] consider a generic semantics of a probabilistic language and apply several forms of abstract interpretation to it.

Borgström et al. [2013] give denotational semantics to Fun, a functional probabilistic language with discrete and continuous distributions and observations, as measure transformers—functions from finite measures to finite measures. They define the semantics of open programs compositionally using arrow-like [Hughes, 1998] combinators. Intuitively, when applied to a measure on the free variables in an expression, the semantics of this expression returns a joint measure on the free variables and the return value. The semantics supports zero-probability observations. However, the core language supports neither higher-order functions nor recursion.

Bhat et al. [2013] presents a reduction system deriving densities of outcomes of first-order probabilistic programs, together with a type system guaranteeing the existence of a density.

Toronto presents a new approach to probabilistic language semantics, which he called “Running probabilistic programs backwards” [Toronto, 2014, Toronto et al., 2015]. He treats a probabilistic program as a deterministic function from a source of randomness (specifically, an infinite tree of values in the set $[0, 1]$) with an associated probability measure P to the set of program outcomes, and defines the semantics of the program as a composition of P and the preimage of f . He then defines the preimage operator compositionally, using arrows [Hughes, 1998]. His approach has the interesting property that it leads directly to an inference algorithm: while preimages are uncomputable in general, they can be approximated by sampling. Toronto defines an implementable abstract semantics and proves that it is a conservative approximation of the exact preimage semantics. However, this semantics is only defined for a first-order language, due to the difficulties with making the higher-order “apply” function measurable [Aumann, 1961].

Ehrhard et al. [2014] present a semantics of a probabilistic extension of PCF with

discrete random draws, based on coherence spaces, and show a full abstraction result for the language with respect to this semantics.

Ścibior et al. [2015] define a denotational sampling-based semantics for a higher-order probabilistic language based on Haskell, in terms of a limit of integrals of a density on traces with respect to Borel measures on traces of finite length. Their semantics is restricted to expressions of simple top-level types, excluding function types and recursive types.

Bizjak and Birkedal [2015] define a step-indexed logical relation for reasoning about equivalence of higher-order probabilistic programs with discrete random draws.

Huang and Morrisett [2016] present a denotational semantics of a first-order language based on computable metric spaces. They restrict the semantics to computable functions and show that the semantics is directly implementable.

Staton et al. [2016] define a denotational and operational semantics for a higher-order typed language with both discrete and continuous random variables and soft and hard conditioning. The authors define a denotational semantics for a first-order language, as a distribution on pairs of program outcomes and scores which can be normalised to yield a distribution on outcomes, and then use the Yoneda embedding of the category of measurable spaces to lift the semantics to the higher-order language. An operational semantics is also defined, and the denotational semantics is shown sound with respect to the operational one. The semantics model used by [Staton et al., 2016] does not support higher-order recursion, only first-order recursion.

Heunen et al. [2017] improve upon that work by providing an alternative, simpler semantics for a higher-order typed language, based on so-called quasi-Borel spaces. The semantics is defined in terms of a category in which the objects are quasi-Borel spaces, that is Borel spaces paired with collections of functions satisfying certain properties, and morphism must preserve these properties when composed with functions from such collections. This semantics also supports only first-order recursion.

2.3 Correctness of inference in probabilistic programs

To our best knowledge, the only other attempts at formalizing and proving correct a trace-based sampling algorithm for probabilistic programs are the recent works by Hur et al. [2015] and Cai [2016].

2.3.1 Multi-site MH for procedural programs [Hur et al., 2015]

Hur et al. [2015] present the first proof of correctness of Metropolis-Hastings for a probabilistic language.

The authors define a while-language with continuous and discrete random draws and observations. They define two different semantics for their language. The first one is a denotational semantics, which defines the “meaning” of a program as the expectation of an arbitrary function applied to the final state of the program—this is equivalent to defining a probability measure on the final states.

The second semantics is a pseudo-deterministic, big-step sampling-based operational semantics. It takes a program, an initial state and a map storing a list of values for every variable (one value for every execution of a probabilistic assignment to the given variable) and returns an updated state together with a weight, which is the product of probability density functions corresponding to the distributions used in the program, applied to the corresponding values in the map.

It is important that the “random vector” maps variables to lists, rather than simple values, because in an imperative language different executions of the same probabilistic assignment represent distinct random choices—a point missed in the design of other algorithms, which, as the authors demonstrate, led to incorrect inference results.

In every step, the MCMC inference algorithm defined by Hur et al. perturbs *every* random variable in the trace, according to a proposal distribution based on the target distribution of the variable and, if available, its previous value. This is different from the algorithm used by Church, where part of the trace is left unchanged. Resampling all the variables makes it possible to define the proposal kernel as an integral of the joint proposal density for all variables in the new trace, avoiding the problem that fixed variables automatically set the value of the integral to zero.

The authors prove that their algorithm satisfies the detailed balance equation for Metropolis-Hastings. They first show that the denotational and operational semantics are equivalent. Then, they observe that the density of the target distribution on traces corresponds to the sampling based semantics. Then they define joint proposal distribution in terms of densities of individual random choices, and use it to show that the acceptance ratio used in the algorithm is computed according to the definition of Metropolis-Hastings.

The authors do not discuss aperiodicity and irreversibility of their algorithm.

2.3.2 Single-site MH for abstract programs [Cai, 2016]

Cai [2016] presents the first correctness proof for density-less trace MCMC, updating one variable at the time. Instead of working with a particular programming language, he makes the proof completely abstract, treating programs as eidetic processes (sequences of probability kernels in which a kernel can depend on values drawn from all previous kernels) and intentionally avoiding dealing with semantics. Hence, his proof is parametric on a mapping from actual programs to eidetic processes and a naming scheme identifying variables in a program, which must satisfy certain properties (most notably prefix-freedom).

This work, not yet published, was only made known to us after the submission of the paper on which chapters 6 and 7 are based.

Chapter 3

Preliminaries

We recapitulate here the less standard background knowledge needed to understand this dissertation, with the aim of making it accessible to readers from various scientific communities. We also present a non-standard stock measure on *program traces*, treated as lists of reals of arbitrary length, which will be used in the semantics of Tabular in Chapter 4 and the probabilistic lambda-calculus in Chapter 6.

We assume the reader already has some elementary knowledge of type systems, lambda calculus and operational semantics. Should it not be the case, there are many useful introductory resources on these subjects, including [Cardelli, 1997], [Barendregt, 1992] and [Winskel, 1993, Chapter 2].

Basic Notation We write $f : X \rightarrow Y$ if f is a function with domain X and codomain Y . If $B \subseteq Y$, we write $f^{-1}(B)$ for the *preimage* of f under B :

$$f^{-1}(B) = \{x \in X \mid f(x) \in B\}$$

We denote by $[x \in B]$ (so-called Iverson bracket) the indicator function of B applied to x , i.e. 1 if $x \in B$ and 0 otherwise:

$$[x \in B] = \begin{cases} 1 & \text{if } x \in B \\ 0 & \text{otherwise} \end{cases}$$

3.1 Probabilistic Inference

We begin by explaining some basic terms related to machine learning, and Bayesian probabilistic modelling in particular. A more comprehensive explanation of these concepts can be found in machine learning textbooks, including [MacKay, 2003, Murphy, 2012, Barber, 2012].

Suppose that we have some observed data D and an unknown parameter θ (possibly being a vector containing several individual parameters), which we want to infer. A *prior* distribution $p(\theta)$ is the probability distribution on θ which models our belief about the likeliness of the parameter θ admitting given values in the absence of any observations.

A *likelihood* $P(D|\theta)$ is the probability that the dataset D will be generated by the model for the given value of the parameter θ .

A *posterior* distribution $p(\theta|D)$ is the probability distribution on θ conditioned on the observed data D . Computation of the posterior distribution is usually the goal of probabilistic inference.

If $\theta = (\theta_1, \dots, \theta_n)$ and $1 \leq i \leq n$, the *marginal* posterior distribution $p(\theta_i|D)$ is the posterior distribution of θ_i given the observed data D .

3.2 Measure Theory

Semantics of probabilistic computations with continuous random variables is usually formulated in terms of *measure-theoretic probability*, which is more general than the straightforward textbook approaches based on Riemann integration of density functions. In this section, we present the basic background knowledge on measure theory needed to understand some parts of this dissertation and a convenient measure space on program traces, useful in defining semantics of probabilistic programs.

A more complete, tutorial-style introduction to measure theory can be found in one of the standard textbooks, for example [Billingsley, 1995].

3.2.1 Basic Measure Theory

This subsection presents some elementary definitions from measure theory, which are needed to understand Section 4.5 in Chapter 4, Section 6.3 of Chapter 6 and Sections 7.2 and 7.3 of Chapter 7.

Measurable Spaces To work with measure theory, we first need to define a *measurable space*.

A σ -*algebra* on a set Ω is a set Σ of subsets of Ω satisfying the following properties:

- $\emptyset \in \Sigma$
- Σ is closed under complements—that is, if $A \in \Sigma$, then $\Omega \setminus A \in \Sigma$.

- Σ is closed under countable unions—that is, if $A_i \in \Sigma$ for all $i \in \mathbb{N}$, then $\bigcup_{i \in \mathbb{N}} A_i \in \Sigma$

If $A \in \Sigma$, then A is called a *measurable subset* of Ω (or simply a *measurable set*, if the context is clear). If Σ is a σ -algebra on Ω , the tuple (Ω, Σ) is called a *measurable space*.

A σ -algebra on Ω *generated by* a set $S \subseteq P(\Omega)$ of subsets of Ω , denoted $\sigma(S)$, is the smallest σ -algebra on Ω containing S . Equivalently

$$\sigma(S) = \bigcap \{ \Sigma \mid S \subseteq \Sigma, \Sigma \text{ is a } \sigma \text{ algebra on } \Omega \}$$

Products of σ -algebras If $(X_1, \Sigma_1), \dots, (X_n, \Sigma_n)$ are measurable spaces, the *product* of $\Sigma_1, \dots, \Sigma_n$ is the σ algebra on $X_1 \times \dots \times X_n$ defined by:

$$\Sigma_1 \otimes \dots \otimes \Sigma_n = \sigma(\{A_1 \times \dots \times A_n \mid A_i \in \Sigma_i \forall i \in 1..n\})$$

Borel σ -algebra on \mathbb{R}^n The *Borel σ -algebra* \mathcal{B} on \mathbb{R} is the σ -algebra generated by the set of open intervals $\{(a, \infty) \mid a \in \mathbb{R}\}$. The Borel σ -algebra \mathcal{B}_n on \mathbb{R}^n is the n -fold closure of \mathcal{B} (that is, $\mathcal{B} \times \dots \times \mathcal{B}$, where we take the product of n copies of \mathcal{B}).

Countably Generated σ -Algebras A σ algebra Σ is *countably generated* if it is generated by a finite set.

Measures If (Ω, Σ) is a measurable space, a *measure* on (Ω, Σ) is a function $\mu : [0, \infty]$ such that:

- $\mu(\emptyset) = 0$
- μ is countably additive—that is, if $A_1, A_2, \dots \in \Sigma$, then $\mu(\bigcup_{i \in \mathbb{N}} A_i) = \sum_{i \in \mathbb{N}} \mu(A_i)$

Note that the value of μ on a measurable set can be infinite. We call the triple (Ω, Σ, μ) a *measure space*.

Products of Measures If μ_1, \dots, μ_n are measures on $(X_1, \Sigma_1), \dots, (X_n, \Sigma_n)$ respectively, the *product* $\mu_1 \otimes \dots \otimes \mu_n$ of μ_1, \dots, μ_n is the unique measure on $(X_1 \times \dots \times X_n, \Sigma_1 \otimes \dots \otimes \Sigma_n)$ satisfying:

$$(\mu_1 \otimes \dots \otimes \mu_n)(A_1 \times \dots \times A_n) = \mu_1(A_1) \dots \mu_n(A_n)$$

for all $A_1 \in \Sigma_1, \dots, A_n \in \Sigma_n$.

Lebesgue measure on $(\mathbb{R}^n, \mathcal{B}_n)$ The *Lebesgue measure* on $(\mathbb{R}, \mathcal{B})$ is the unique measure λ such that $\lambda([a, b]) = b - a$ for all $a, b \in \mathbb{R}, a \leq b$. The Lebesgue measure λ_n on $(\mathbb{R}^n, \mathcal{B}_n)$ is the n -fold closure of λ —that is, $\lambda \times \cdots \times \lambda$, where λ appears n times.

Probability Measures A measure μ on (Ω, Σ) is called a *probability measure* if $\mu(\Omega) = 1$ and *subprobability measure* if $\mu(\Omega) \leq 1$.

σ -Finite Measures A measure μ on (Ω, Σ) is *σ -finite* if there exists a sequence of measurable sets A_i such that $A_i \subseteq A_{i+1}$ for all i and $\Omega = \bigcup_{i \in \mathbb{N}} A_i$ and $\mu(A_i) < \infty$ for all i .

Measurable Functions A function f between measurable spaces (X, Σ_X) and (Y, Σ_Y) is *measurable* Σ_X/Σ_Y if for all $B \in \Sigma_Y$, $f^{-1}(B) \in \Sigma_X$. We write simply that f is measurable if the σ -algebras are clear from the context.

Lebesgue Integral If (X, Σ_X) is a measurable space, a measurable function $f : X \rightarrow [0, \infty)$ is called *simple* if its image is a finite set. Every simple function f can be expressed as:

$$f(x) = \sum_{i=1}^n \alpha_i [x \in A_i]$$

where $A_i = f^{-1}(\alpha_i)$.

The Lebesgue integral $\int f(x) \mu(dx)$ of the simple function $f : X \rightarrow [0, \infty)$ such that $f(x) = \sum_{i=1}^n \alpha_i [x \in A_i]$ with respect to the measure μ on (X, Σ_X) is defined as:

$$\int f(x) \mu(dx) = \sum_{i=1}^n \alpha_i \mu(A_i)$$

Every non-negative measurable function can be approximated to arbitrary precision by simple functions. The Lebesgue integral of an arbitrary non-negative measurable function from X to \mathbb{R} is defined as:

$$\int f(x) \mu(dx) = \sup \left\{ \int g(x) \mu(dx) \mid g \text{ simple, } g \leq f \right\}$$

where the inequality $g \leq f$ is defined pointwise. We write the above integral simply as $\int f(x) dx$ if the measure with respect to which the function is integrated is clear from the context.

The restricted integral $\int_A f(x)\mu(dx)$ is defined to be:

$$\int_A f(x)\mu(dx) = \int f(x)[x \in A]\mu(dx)$$

If (X, Σ_X, μ) is a measure space and $f : X \rightarrow [0, \infty)$ is measurable, the function $\nu(A) = \int_A f(x)\mu(dx)$ is a measure on (X, Σ_X) . We call the function f the *density* of ν . The integral $\int_A f(x)\mu(dx)$ is *absolutely continuous* with respect to μ —that is, if $\mu(A) = 0$, then $\int_A f(x)\mu(dx) = 0$.

σ -algebra Restriction The *restriction* $\Sigma|_A$ of a σ -algebra Σ on Ω is defined to be

$$\Sigma|_A = \{B \cap A \mid B \in \Sigma\}$$

Then $(A, \Sigma|_A)$ is a measurable space.

Measure Restriction If (Ω, Σ, μ) is a measure space, we define $\mu|_A$ to be a restriction of the measure μ to A :

$$\mu|_A(B) = \mu(A \cap B)$$

The restriction $\mu|_A$ is a measure on (Ω, Σ) .

Continuous Probability Distributions A *continuous probability distribution* D , typically parametrised by some parameters, is a probability measure on $(\mathbb{R}, \mathcal{B})$. We call pdf_D the *density* of D (or, in full, the *probability density function* of D), if pdf_D is the density of D with respect to the Lebesgue measure λ on $(\mathbb{R}, \mathcal{B})$ —that is, $D(A) = \int_A \text{pdf}_D(x)\lambda(dx)$.

Variational Norm The *variational norm* $\|\nu_1 - \nu_2\|$ of two measures ν_1 and ν_2 on (Ω, Σ) is defined to be $\|\nu_1 - \nu_2\| = \sup_{A \in \Sigma} |\nu_1(A) - \nu_2(A)|$.

3.2.2 A Measure Space on Program Traces

In this section, we present a measurable space of program traces (treated as real-valued lists of arbitrary length) and a stock measure on this space of traces, with respect to which density functions will be integrated to give semantics to programs in Chapter 4 and 6. A basic knowledge of the existence of this measure space is necessary to understand Section 4.5 of Chapter 4, where it is used to define marginals distributions

of random columns in Tabular. A more in-depth understanding is required to understand Section 6.3 of Chapter 6 and Chapter 7, and especially the detailed proofs of lemmas and theorems in these chapters, included in Appendix E.

We begin by defining a measurable space $(\mathbb{U}, \mathcal{S})$ of *program traces*. As a program trace in a general-purpose probabilistic language can be any sequence (possibly empty) of real values of arbitrary length, the space \mathbb{U} of program traces is the disjoint union of n -fold Cartesian products of real values for any n , that is, $\mathbb{U} \triangleq \bigsqcup_{n \in \mathbb{N}} \mathbb{R}^n$ (note that \mathbb{N} includes 0).

We want \mathcal{S} to be a generalization of the Borel σ -algebra on the space of fixed-dimensional tuples (that is, \mathbb{R}^n) to the space of sequences of arbitrary length. Let \mathcal{B}_n be the Borel σ -algebra on \mathbb{R}^n for $n \geq 0$, where $\mathcal{B}_0 = \{\{\}, \{\{\}\}\}$ and $\mathbb{R}^0 = \{\{\}\}$. We define \mathcal{S} as follows:

$$\mathcal{S} \triangleq \left\{ \bigsqcup_{n \in \mathbb{N}} H_n \mid H_n \in \mathcal{B}_n \text{ for all } n \in \mathbb{N} \right\}$$

That is, each set in \mathcal{S} is a countable disjoint union of Borel subsets of \mathbb{R}^n for each $n \in \mathbb{N}$. We can easily verify that \mathcal{S} is indeed a σ -algebra:

Lemma 1 \mathcal{S} is a σ -algebra on \mathbb{U} .

Proof: We need to check that $\mathbb{U} \in \mathcal{S}$ and that \mathcal{S} is closed under complement and countable union.

- We have $\mathbb{U} = \bigsqcup_{n \in \mathbb{N}} \mathbb{R}^n$, so obviously $\mathbb{U} \in \mathcal{S}$
- Let $A \in \mathcal{S}$. Then $A = \bigsqcup_{n \in \mathbb{N}} H_n$, where $H_n \in \mathcal{B}_n$ for all n . We have $\mathbb{U} \setminus A = \bigsqcup_{n \in \mathbb{N}} (\mathbb{R}^n \setminus H_n)$. Each Borel σ -algebra \mathcal{B}_n is by definition closed under complement, so $(\mathbb{R}^n \setminus H_n) \in \mathcal{B}_n$ for all $n \in \mathbb{N}$. Thus, $\mathbb{U} \setminus A \in \mathcal{S}$, and so \mathcal{S} is closed under complement.
- Let $A_i \in \mathcal{S}$ for all $i \in \mathbb{N}$. Then for each i , $A_i = \bigsqcup_{n \in \mathbb{N}} H_{in}$, where $H_{in} \in \mathcal{B}_n$. Each \mathcal{B}_n is closed under countable union, so for all n , $\bigcup_{i \in \mathbb{N}} H_{in} \in \mathcal{B}_n$.

We have $\bigcup_{i \in \mathbb{N}} A_i = \bigcup_{i \in \mathbb{N}} (\bigsqcup_{n \in \mathbb{N}} H_{in}) = \bigsqcup_{n \in \mathbb{N}} (\bigcup_{i \in \mathbb{N}} H_{in})$, so $\bigcup_{i \in \mathbb{N}} A_i \in \mathcal{S}$. Hence \mathcal{S} is closed under countable union.

■

Hence, $(\mathbb{U}, \mathcal{S})$ is a measurable space.

We can also show that \mathcal{S} is countably generated, which will be useful in Chapter 7. It is a well-known fact that for each $n \in \mathbb{N}$, the Borel σ -algebra \mathcal{B}_n on \mathbb{R}^n is countably generated. Let $T_0 = \{\{\}, \{\square\}\}$ for each $n > 0$, let T_n be a countable subset of \mathcal{B}_n such that $\mathcal{B}_n = \sigma(T_n)$.

Lemma 2 $\mathcal{S} = \sigma(\biguplus_{n \in \mathbb{N}} T_n)$

Proof: We need to show that $\mathcal{S} \subseteq \sigma(\biguplus_{n \in \mathbb{N}} T_n)$ and $\mathcal{S} \supseteq \sigma(\biguplus_{n \in \mathbb{N}} T_n)$:

- $\mathcal{S} \supseteq \sigma(\biguplus_{n \in \mathbb{N}} T_n)$: Take any $A \in \biguplus_{n \in \mathbb{N}} T_n$. Then $A \in T_i$ for some $i \in \mathbb{N}$. Thus, A can be represented as $\biguplus_{n \in \mathbb{N}} H_n$, where $H_i \in T_i \subseteq \mathcal{B}_i$ and $H_j = \emptyset \subseteq \mathcal{B}_j$ for $j \neq i$. Hence, $A \in \mathcal{S}$. Thus, $\biguplus_{n \in \mathbb{N}} T_n \subseteq \mathcal{S}$. Since \mathcal{S} is a σ -algebra containing $\biguplus_{n \in \mathbb{N}} T_n$ and $\sigma(\biguplus_{n \in \mathbb{N}} T_n)$ is, by definition, the smallest σ -algebra containing $\biguplus_{n \in \mathbb{N}} T_n$, we have $\sigma(\biguplus_{n \in \mathbb{N}} T_n) \subseteq \mathcal{S}$, as required.
- $\mathcal{S} \subseteq \sigma(\biguplus_{n \in \mathbb{N}} T_n)$: Let $H = \biguplus_{n \in \mathbb{N}} H_n \in \mathcal{S}$. Then for each $i \in \mathbb{N}$, $H_i \in \mathcal{B}_i = \sigma(T_i) \subseteq \sigma(\biguplus_{n \in \mathbb{N}} T_n)$. Thus, since a σ -algebra is closed under countable union, we have $H = \biguplus_{n \in \mathbb{N}} H_n \in \sigma(\biguplus_{n \in \mathbb{N}} T_n)$. Therefore, $\mathcal{S} \subseteq \sigma(\biguplus_{n \in \mathbb{N}} T_n)$, as required. ■

Lemma 3 The set $\biguplus_{i \in \mathbb{N}} T_i$ is countable.

Proof: For any $i \in \mathbb{N}$, T_i is countable by definition. Thus, $\biguplus_{i \in \mathbb{N}} T_i$ is a countable union of countable sets, and so it is countable. ■

Corollary 1 The σ -algebra \mathcal{S} is countably generated.

In order to define a distribution on program traces, we need a *stock measure* on program traces, generalising the Lebesgue measure to the measurable space $(\mathbb{U}, \mathcal{S})$, with respect to which the trace density function will be integrated. This measure μ is defined as follows:

$$\mu(A) \triangleq \sum_{n \in \mathbb{N}} \lambda_n(A|_{\mathbb{R}^n})$$

where λ_n is the Lebesgue measure on $(\mathbb{R}^n, \mathcal{B}_n)$ for $n \geq 1$ and $\lambda_0 = \delta(\square)$.

Lemma 4 μ is a measure on $(\mathbb{U}, \mathcal{S})$

Proof: We need to check that $\mu(A) \in [0, \infty]$ for all $A \in \mathcal{S}$, that $\mu(\emptyset) = 0$ and that μ is countably additive:

- For all $A \in \mathcal{S}$, we have $\lambda_n(A|_{\mathbb{R}^n}) \in [0, \infty]$ by a property of the Lebesgue measure, so obviously $\mu(A) = \sum_{n \in \mathbb{N}} \lambda_n(A|_{\mathbb{R}^n}) \in [0, \infty]$
- If $A = \emptyset$, then for every $n \in \mathbb{N}$, $A|_{\mathbb{R}^n} = \emptyset$, and since λ_n is a measure, $\lambda_n(\emptyset) = 0$. Hence, $\mu(\emptyset) = 0$.
- Let $A_m \in \mathcal{S}$ for all $m \in \mathbb{N}$. Then for every m , $A_m = \sum_{n \in \mathbb{N}} H_{mn}$, where $H_{mn} \in \mathcal{B}_n$ for every n . We have:

$$\begin{aligned} \mu\left(\sum_{m \in \mathbb{N}} A_m\right) &= \mu\left(\sum_{m \in \mathbb{N}} \left(\sum_{n \in \mathbb{N}} H_{mn}\right)\right) = \mu\left(\sum_{n \in \mathbb{N}} \left(\sum_{m \in \mathbb{N}} H_{mn}\right)\right) = \sum_{n \in \mathbb{N}} \lambda_n\left(\sum_{m \in \mathbb{N}} H_{mn}\right) \\ &= \sum_{n \in \mathbb{N}} \sum_{m \in \mathbb{N}} \lambda_n(H_{mn}) = \sum_{m \in \mathbb{N}} \sum_{n \in \mathbb{N}} \lambda_n(H_{mn}) = \sum_{m \in \mathbb{N}} \mu(A_m) \end{aligned}$$

as required. The equality $\sum_{n \in \mathbb{N}} \sum_{m \in \mathbb{N}} \lambda_n(H_{mn}) = \sum_{m \in \mathbb{N}} \sum_{n \in \mathbb{N}} \lambda_n(H_{mn})$ follows from Tonelli's theorem for series [Tao, 2011].

■

We have shown that $(\mathbb{U}, \mathcal{S}, \mu)$ is a *measure space*. We also want to prove that the measure μ is σ -finite. We require this property to use some standard results from measure theory literature later in this chapters. Below, we write $[a, b]^n$ for $\{(x_1, \dots, x_n) \mid x_i \in [a, b] \ \forall i \in 1..n\}$ if $n \geq 1$ and $[a, b]^0$ for $\{\emptyset\}$.

Lemma 5 *The measure μ is σ -finite.*

Proof: We have $\mathbb{R}^n = \bigcup_{k \in \mathbb{N}} [-k, k]^n$ for all $n \in \mathbb{N}$. Thus, $\mathbb{U} = \bigcup_{n \in \mathbb{N}} \mathbb{R}^n = \bigcup_{n \in \mathbb{N}} \bigcup_{k \in \mathbb{N}} [-k, k]^n$. Thus, \mathbb{U} is a union of countably many sets of the form $[-k, k]^n$. Meanwhile, for all $k, n \in \mathbb{N}$, $\mu([-k, k]^n) = \lambda_n([-k, k]^n) = (2k)^n < \infty$. Hence, \mathbb{U} is a union of countably many sets on which μ is finite, so μ is a σ -finite measure.

■

In the rest of this dissertation, we use $\int f(s) ds$ as an abbreviation of the integral $\int f(s) \mu(ds)$ of f with respect to the measure μ on traces. We will also write $|s|$ for the length of s and $s@t$ for the concatenation of the sequences s and t . We write s_i for the i -th element of s (starting from 1) and $s_{i..j}$ for the subtrace $[s_i, \dots, s_j]$ of the trace $s = [t_1, \dots, t_n]$ (where $1 \leq i \leq j \leq n$).

3.2.3 Metric and Topological Spaces

This section presents some basic definitions concerning metric and topological spaces, which can give rise to measurable spaces. It also presents an alternative definition of the measurable space of program traces, as a space induced by a metric. The definitions presented here are used in Section 6.3 of Chapter 6 and in Chapter 7, but an in-depth understanding of metric and topological spaces is only needed to understand the proofs in Appendix E.

Borel σ -Algebra Induced by a Metric A *metric space* is a pair (Ω, d) , where Ω is a set and d is a real valued function on $\Omega \times \Omega$ which satisfies $d(x, x) = 0$ and $d(x, y) + d(y, z) \geq d(x, z)$ for all $x, y, z \in \Omega$. A subset A of Ω is called *open* if every point x in A has a neighbourhood lying completely in A —that is, there exists $\varepsilon > 0$ such that $\{y \in \Omega \mid d(x, y) < \varepsilon\} \subseteq A$.

A *topology* on a set Ω is the set \mathcal{O} of subsets of Ω such that:

- $\emptyset \in \mathcal{O}$ and $\Omega \in \mathcal{O}$
- \mathcal{O} is closed under finite intersections—that is, if $O_1, \dots, O_n \in \mathcal{O}$, then $O_1 \cap \dots \cap O_n \in \mathcal{O}$
- \mathcal{O} is closed under countable unions—that is, if $O_i \in \mathcal{O}$ for all $i \in \mathbb{N}$, then $\bigcup_{i \in \mathbb{N}} O_i \in \mathcal{O}$.

The elements of \mathcal{O} are called *open sets* and the pair (Ω, \mathcal{O}) is called a *topological space*.

A topology on Ω *induced* by a metric d is the smallest topology containing all the open sets of (Ω, d) .

The σ -algebra $\mathcal{B}(\Omega)$ on Ω generated by the topology \mathcal{O} is the *Borel σ -algebra*. We call the σ -algebra on Ω generated by the topology induced by d the *Borel σ -algebra induced by d* .

Closed Subsets of Metric and Topological Spaces Given a sequence of points x_n in a metric space (X, d) , we say that x is the *limit* of x_n , written $x_n \rightarrow x$, if for all $\varepsilon > 0$, there exists an N such that $d(x_n, x) < \varepsilon$. A subset A of a metric space is *closed* if it contains all the limit points—that is, if $x_n \in A$ for all n and $x_n \rightarrow x$, then $x \in A$.

A subset A of a topological space (X, \mathcal{O}) is *closed* if its complement $X \setminus A$ is open. If (X, \mathcal{O}) is induced by a metric space (X, d) , then the closed sets in (X, \mathcal{O}) are precisely the closed sets in (X, d) .

A σ algebra induced by a metric space contains all its open and closed sets.

Separable Metric Spaces A subset A of a metric space (X, d) is *dense* if

$$\forall x \in X, \varepsilon > 0 \exists y \in A \quad d(x, y) < \varepsilon$$

A metric space is *separable* if it has a countable dense subset.

Continuous and Measurable Functions If (X, d_X) and (Y, d_Y) are metric spaces, a function $f : X \rightarrow Y$ is *continuous* (with respect to the metrics d_X, d_Y) if for every open subset O of (Y, d_Y) , $f^{-1}(O)$ is an open subset of (X, d_X) . Equivalently, f is continuous if for every $x \in X$ and $\varepsilon > 0$, there exists δ such that for every $x' \in X$, if $d_X(x, x') < \delta$, then $d_Y(f(x), f(x')) < \varepsilon$.

If (X, \mathcal{O}_X) and (Y, \mathcal{O}_Y) are topological spaces, a function $f : X \rightarrow Y$ is *continuous* (with respect to the topologies $\mathcal{O}_X, \mathcal{O}_Y$) if for every open subset O of (Y, d_Y) , $f^{-1}(O)$ is an open subset of (X, d_X) . Hence, a continuous function between metric spaces is continuous with respect to the topologies induced by them.

If Σ_X and Σ_Y are Borel σ -algebras generated by topologies \mathcal{O}_X and \mathcal{O}_Y , respectively, and $f : X \rightarrow Y$ is continuous with respect to \mathcal{O}_X and \mathcal{O}_Y , then f is measurable Σ_X/Σ_Y . In this case, we call the function f *Borel-measurable*.

Products in Metric and Topological Spaces The Manhattan metric of metric spaces $(X_1, d_1), \dots, (X_n, d_n)$ is the metric d on $X_1 \times \dots \times X_n$ defined by $d((x_1, \dots, x_n), (y_1, \dots, y_n)) = d_1(x_1, y_1) + \dots + d_n(x_n, y_n)$.

A product of topological spaces $(X_1, \mathcal{O}_1), \dots, (X_n, \mathcal{O}_n)$ is the smallest topology on $X_1 \times \dots \times X_n$ such that all the maps π_i (where $\pi_i(x_1, \dots, x_n) = x_i$) are continuous.

If (X_i, \mathcal{O}_i) is the topology induced by separable metric spaces (X_i, d_i) for each $i \in 1..n$, the topology induced by the Manhattan metric of $(X_1, d_1), \dots, (X_n, d_n)$ is the product of $(X_1, \mathcal{O}_1), \dots, (X_n, \mathcal{O}_n)$.

Alternative definition of the measurable space of program traces A convenient way of showing that a function is measurable is by showing that it is continuous as a function between metric spaces. Because of that, it is convenient to redefine the σ -algebra \mathcal{S} as an algebra induced by a metric on \mathbb{U} . That is, we want this metric to

induce a topology on \mathbb{U} whose open sets generate \mathcal{S} . We can define this metric as follows:

$$d(s, s') \triangleq \begin{cases} \sum_{i=1}^{|s|} |s_i - s'_i| & \text{if } |s| = |s'| \\ \infty & \text{otherwise} \end{cases}$$

where $|s|$ denotes the length of s , as defined before. The set of open sets in this metric space is clearly the disjoint union of open sets $\mathcal{O}(\mathbb{R}^n, d_n)$ of (\mathbb{R}^n, d_n) for each $n \in \mathbb{N}$, where d_n is the standard Manhattan metric on \mathbb{R}^n .

Lemma 6 *The σ algebra \mathcal{S} on \mathbb{U} is induced by the metric d .*

Proof: Let $A \in \mathcal{S}$. Then $A = \biguplus_{n \in \mathbb{N}} H_n$, where $H_n \in \mathcal{B}_n$ for each n . As each \mathcal{B}_n is generated by $\mathcal{O}(\mathbb{R}^n, d_n)$ and $\mathcal{O}(\mathbb{R}^n, d_n) \subseteq \mathcal{O}(\mathbb{U}, d)$, we have $H_n \in \sigma(\mathcal{O}(\mathbb{U}, d))$, so $A \in \sigma(\mathcal{O}(\mathbb{U}, d))$, as σ -algebras are closed under countable union. ■

3.2.4 Subprobability and Probability Kernels

The following definitions are only used in Chapter 7.

If (X, Σ_X) and (Y, Σ_Y) are measurable spaces, a function $K : X \times \Sigma_Y \rightarrow [0, 1]$ is called a *subprobability kernel* from (X, Σ_X) to (Y, Σ_Y) if:

- For every $B \in \Sigma_Y$, the function $K(\cdot, B)$ is measurable $\Sigma_X / \mathcal{B}(\mathbb{R})|_{[0,1]}$
- For every $x \in X$, the function $K(x, \cdot)$ is a subprobability measure.

Similarly, K is a *probability kernel* if $K(x, \cdot)$ is a probability measure for every x .

If $K(x, B) = \int_B k(x, y) \mu(dy)$, we say that K has *density* k with respect to the measure μ .

3.3 Metropolis-Hastings Sampling in General State Spaces

This part of the chapter describes the Metropolis-Hastings (MH) sampling algorithm. The material in this section is only needed to understand Chapter 7.

For more background theory on Markov chains on general state spaces, consult [Nummelin, 1984]. A more gentle introduction to Metropolis-Hastings, focusing more

on its practical significance, can be found in one of the many machine learning textbooks, for example [MacKay, 2003] or [Murphy, 2012]. Meanwhile, for a more comprehensive account of the theory of MH on general measurable spaces, refer to [Tierney, 1994], [Tierney, 1998] and [Roberts et al., 2004].

Sampling Algorithms The goal of Bayesian inference is to compute the *posterior* distribution $p(\mathbf{x}|D)$ of some random function (the *target distribution*), given the *prior* distribution $p(\mathbf{x})$ and a set of *observations* D . In most cases, the posterior distribution $p(\mathbf{x}|D)$ is impossible to compute analytically and it is necessary to resort to approximations. *Sampling algorithms* generate large numbers of samples from the posterior distribution. These samples can then be used to compute the (approximate) properties of the posterior distribution, such as its mean and variance, or to compute the approximate expectation of some function of the model output.

Random Variables In a measure-theoretic setting, a *random variable* on a set Y is a measurable function $f : \Omega \rightarrow Y$. This function can be understood as a map from a source of randomness ω (being an element of the set Ω) to an observable outcome $f(\omega)$ of a random draw.

3.3.1 Markov Chains

A *Markov chain* on a measurable space (X, Σ) is a sequence of random variables on X such that the distribution of each variable depends only on the value of the previous variable. A Markov chain is defined by a distribution on the initial value X_0 (often assumed to be fixed) and a probability kernel P (from (X, Σ) to (X, Σ)) such that $P(X_n, A)$ is the probability that $X_{n+1} \in A$.

We say that a probability measure π on (X, Σ) is the *invariant distribution* (or *stationary distribution*) of the Markov chain defined by P if

$$\pi(A) = \int_A P(x, A) \pi(dx)$$

for all $A \in \Sigma$.

The probability $P^n(x, B)$ that the n -th element X_n of the Markov chain is in B if the first element was x can be expressed inductively as follows:

$$\begin{aligned} P^0(x, B) &= [x \in B] \\ P^{n+1}(x, B) &= \int P^n(y, B) P(x, dy) \end{aligned}$$

Similarly, the probability $\underline{P}^n(x, B)$ that *all* first n elements of the Markov chain P (as well as the starting element x) are in B can be expressed in the following way:

$$\begin{aligned}\underline{P}^0(x, B) &= [x \in B] \\ \underline{P}^{n+1}(x, B) &= \int_B \underline{P}^n(y, B) P(x, dy)\end{aligned}$$

Properties of Markov Chains Let P be a Markov chain on (X, Σ) with stationary distribution π and let ϕ be any measure on (X, Σ) . We say that the kernel P is ϕ -*irreducible* if $\phi(X) > 0$ and for every $x \in X$ and $A \in \Sigma$ such that $\phi(A) > 0$, there exists $n > 0$ such that $P^n(x, A) > 0$. The kernel is *strongly ϕ -irreducible* if the above holds for $n = 1$.

A ϕ -irreducible kernel P is *periodic* if there exist $d > 1$ and disjoint sets $A_1, \dots, A_d \in \Sigma$ such that $\pi(A_i) > 0$ for all $i \in 1..d$ and for every i , if $x \in A_i$, then $P(x, A_{i+1}) = 1$ if $i < d$ and $P(x, A_i) = 1$ if $i = d$. Otherwise, the kernel is *aperiodic*.

A ϕ -irreducible kernel P is *Harris recurrent* if for all $x \in X$ and all sets $A \in \Sigma$ such that $\pi(A) > 0$, the probability that the Markov chain defined by P will reach A in finitely many steps starting from x is 1.

3.3.2 Metropolis-Hastings Markov chain Monte Carlo (MH-MCMC)

Sampling directly from the target (posterior) distribution is often computationally expensive or even practically impossible, because of the presence of conditioning, which sometimes cannot easily be simulated in a purely generative fashion. Because of this, the Metropolis-Hastings MCMC algorithm samples from a simpler *proposal* distribution and subsequently discards some samples to ensure that the distribution of samples matches the posterior.

More precisely, the MCMC algorithm generates samples by performing a random walk on the parameter space, thus constructing a Markov chain whose stationary distribution $\pi(\cdot)$ is the distribution we want to sample from. In the Metropolis-Hastings variant, at each step, the algorithm draws a sample \hat{x} from some *proposal kernel* $Q(x, \cdot)$ (proposal distribution parametrised by X) centred at the current state x of the Markov chain. The sample is subsequently accepted with probability $\alpha(x, \hat{x})$, depending on both the previous and the current sample, such that the resulting procedure generates samples distributed according to the target posterior distribution.

From a generative perspective, the algorithm works as follows:

- (1) Set $k = 0$ and choose some initial value x

- (2) Sample a new value \hat{x} from the proposal kernel $Q(x_k, \cdot)$.
- (3) Accept the sample \hat{x} with probability $\alpha(x_k, \hat{x})$. If accepted, set x_{k+1} to \hat{x} , otherwise set x_{k+1} to x_k .
- (4) Set k to $k + 1$ and continue from step 2.

Proposal Kernel with a Density We now concentrate on the particular case where the proposal kernel and the target distribution have densities with respect to the same measure—that is, $Q(x, A) = \int_A q(x, y) \mu(dy)$ and $\pi(A) = \int_A \hat{\pi}(x) \mu(dx)$, where $q(\cdot, \cdot)$ and $\hat{\pi}(\cdot)$ are densities of the proposal kernel and target distribution, respectively. In this case, the correct acceptance ratio is:

$$\alpha(x, y) = \begin{cases} 0 & \text{if } \hat{\pi}(x)q(x, y) = 0 \\ \frac{\hat{\pi}(y)q(x, y)}{\hat{\pi}(x)q(y, x)} & \text{otherwise} \end{cases}$$

This acceptance ratio ensures that the resulting Markov chain is *reversible*.

The algorithm described above constructs a Markov chain with the following transition kernel:

$$P(x, A) = \int_A \alpha(x, y) q(x, y) \mu(dy) + [x \in A] \int (1 - \alpha(x, y)) q(x, y) \mu(dy)$$

Chapter 4

Tabular: A Schema-based Probabilistic Language

Acknowledgement This chapter is based on the paper “Probabilistic Programs as Spreadsheet Queries” [Gordon et al., 2015] published at the 2015 European Symposium on Programming (ESOP). The paper was joint work with Andrew D. Gordon, Claudio Russo, Johannes Borgström, Nicolas Rolland, Thore Graepel and Daniel Tarlow.

Most existing probabilistic languages are essentially probabilistic extensions of conventional programming languages. They are convenient tools for specifying complex Bayesian models, but require all the necessary data to be loaded and put in the right data structures. This can often be problematic and require a large amount of data pre-processing.

The Tabular language, first presented by Gordon et al. [2014], 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 data do not need to be transformed to conform to the program.

In this chapter, we present the syntax, dependent type system and semantics of a new, substantially enhanced version of Tabular, which features user-defined functions

and queries on inference results. We present a reduction relation reducing Tabular programs with function applications to Core models containing only simple expressions and corresponding directly to factor graphs. By **Theorem 1**, this reduction is type-sound. Taking the view that the meaning of a program is defined by the particular quantities we want to estimate, rather than just marginal distributions on program variables, we define the semantics of a Tabular program to be the database of expected outcomes of queries on probabilistic expressions. **Theorem 2** shows that this semantics is well-defined.

4.1 Introduction and Examples

In this section, we introduce Tabular informally, explaining its features by examples. We also list the contributions of this chapter and compare the current formulation of the language to the preliminary version of Tabular [Gordon et al., 2014].

4.1.1 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 computed.

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 adapted from [Gordon et al., 2014]), implementing the TrueSkill model [Herbrich et al., 2006] 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	real

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:

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.

¹As explained in section 4.2, 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.

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.

Obviously, in order to perform inference in the above model, we need to parametrize it on a particular dataset. In Tabular, like in BUGS and Stan, input data is decoupled from the program and is loaded by the compiler from a separate spreadsheet. 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		Matches			
ID	Name	ID	Player1	Player2	Win1
0	"Alice"	0	0	1	false
1	"Bob"	1	1	2	false
2	"Cynthia"				

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	Player1	Player2	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	Player1	Player2	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)

4.1.2 User-Defined, Dependently-Typed Functions

In addition to basic models, the original formulation of Tabular featured a fixed collection of conjugate models, which could be used to write complex programs more concisely. In the new version, these models are replaced by user-defined functions, which are defined in the same way as ordinary tables. Functions help Tabular users make their schemas shorter and more concise by abstracting away arbitrary repeated blocks of code which only differ by some values used in the model expressions. The language features a library of standard functions, replacing the fixed collection of the preliminary version.

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 $\text{Dirichlet}[2]([1.0, 1.0])$ is just the uniform distribution on pairs of two probabilities adding up to 1, and $\text{Discrete}[2](V)$ 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**[N] indicates that the given column is an array of reals of size determined by the variable N , and **mod**(N) denotes a non-negative integer smaller than N .

It is worth noting that in the definition of `CDiscrete` we could alternatively make the entire pseudocount vector passed to $\text{Dirichlet}[N]$ 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)

The reduction algorithm presented later in this chapter reduces this table to the form presented above, 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. 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)[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.

4.1.3 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 “counts” 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].counts(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].counts(V), Dirichlet[2] denotes the type of distribution from which we want to extract a property, counts 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.

Note that 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 query operator.

4.1.4 Critique of the Preliminary Version

A preliminary version of the Tabular language was presented by Gordon et al. [2014]. While the proposed design went a long way in making probabilistic programming more intuitive and data-centric, the original formulation of the language suffered from some deficiencies and left some room for improvement.

For one thing, the modelling language of the preliminary version of Tabular is rather limited. Each column of data can only be modelled by a simple Fun expression, a model from a small, fixed library of primitive models or an indexed library model. There is no way of creating custom, reusable models and if the user wants to use, for instance, their own conjugate model, they have to copy and paste the columns containing all the parameters and the output of the model wherever they are used, changing the hyperparameter values as required.

Apart from this, the syntax, semantics and type system of the language are rather unintuitive. The type system of original Tabular associates each well-formed schema with a quintuple of nested record types, which give the types of individual parameters, inputs and outputs defined by the schema. Since a model expression in a single column of a table can have its own parameters and outputs, it is not immediately clear what the random variables in the program are. The typing rules split the columns into their parameter and output components, treating different kinds of variables defined by a column separately and inserting their types in different nested record types—hence, determining the type of a schema or a table requires manipulating nested records, making it difficult to understand what the type of a given model should be.

The paper also defines the semantics of Tabular, by means of a reduction relation translating well-typed Tabular schemas to so-called models, consisting of triples of Fun expressions, which have well-defined measure-theoretic semantics [Gordon et al., 2013]. Defining the semantics by means of translation to another language adds a layer of indirection. Moreover, the translation rules are very complex, and rely on manipulating let-spines.

It is also not possible to compute any quantities depending on inference results inside Tabular. This means that, for instance, to perform decision theory, the user has to first compute the marginal distribution of interest in Tabular, and then use another tool to perform any computation based on some property (mean, variance, etc.) of this distribution.

Finally, the original version of Tabular is not embedded in Excel (nor any other spreadsheet package), but implemented as a standalone application based on Microsoft Access. However, the new Excel plugin was implemented by other members of the team and is outside the scope of this dissertation.

4.1.5 Contributions

We present a revised, significantly improved version of Tabular, featuring user-defined functions, dependent types and pseudo-deterministic queries, which were all absent on the preliminary version. This new version also fixes the problems with α -conversion and variable substitution, found in the original formulation of Tabular, by giving each column both a fixed external name and an α -convertible internal name.

We define a new, more structural type system of Tabular, in which the type of any well-defined table has the same form as the table itself, with the function calls expanded and the model expressions removed. In order to catch common modelling errors, we introduce basic dependent expression types: an array type can have a size depending on a previously defined deterministic column, and an integer-valued expression can have a bound, also defined by the value of a previous, deterministic column.

We provide a reduction system reducing Tabular programs with function calls and indexing to schemas in Core Tabular, containing just basic model expressions. Core schemas have the property that each column represents exactly one variable or array of variables (with as many components as there are rows in the given table), so they are easy to understand and have a straightforward interpretation as factor graphs with plates. We prove that every well-typed Tabular table reduces deterministically to a unique Core Tabular table with the same type.

To enable the user to compute derived quantities based on inference results within Tabular, we introduce a new **infer** operator, which extracts a property of an inferred distribution, such as mean or variance. The value returned by **infer** can then be used in subsequent computations, which are performed after inference is completed.

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, we split columns into three spaces: **det**, **rnd** and **qry**; we add space annotations to the column types and extend the type system to ensure that the constraints on dependencies between columns are preserved.

In addition to the reduction relation reducing schemas to Core form, we also define the semantics of Core Tabular in two steps: first, we present a sampling-based op-

erational semantics, which can be integrated to obtain marginal measures on queried expressions. Subsequently, we define another operational semantics, which computes the values of pseudo-deterministic queries, taking the previously computed map of marginals as input. We show that the composition of these two semantics, applied to a well-typed schema and a conformant database, yields a well-formed output database.

In summary, this chapter makes the following novel research contributions:

- A dependent type system for probabilistic database schemas with information flow tracking
- A syntactic embedding of user-defined functions inside schemas
- A reduction system reducing probabilistic schemas to a Core form, equivalent to factor graphs
- **Theorem 1:** A type soundness result for the reduction system
- A sampling-based semantics mapping a source of randomness to the values of queried expressions, which can be integrated to yield marginal measures on queried expressions
- An operational query semantics computing results of pseudo-deterministic queries, given marginal measures of queried expressions.
- **Theorem 2:** The composition of the sampling-based semantics and the query semantics, applied to a well-typed schema with a conformant input database, returns a well-formed output database.

4.1.6 Interface and Implementation

The new version of Tabular is implemented as an Excel plugin and both the database and the annotated schema are loaded directly from spreadsheets. The aim of this embedding was to provide domain experts with a convenient modelling environment. The Tabular implementation is based on the Infer.NET backend, whose main inference algorithm is Expectation Propagation [Minka, 2001].

Both the Excel interface and the backend were designed and developed by other members of the research team and are left outside the scope of this dissertation. For more details, see the original paper [Gordon et al., 2014].

Note About the Inference Engine

The default inference engine of Tabular, which uses Expectation Propagation [Minka, 2001], only allows conjugate models to be used in Tabular programs (that is, models in which the posterior distribution of a variable has the same form as the prior). This means that, for instance, defining a Gaussian distribution whose mean has a Gamma prior will result in an error. However, conjugacy is not enforced by the language design, because it is a requirement of a particular inference algorithm and other backends could be implemented.

4.2 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.

4.2.1 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 $\tau_i = [c_i \mapsto a_i^{j \in 1..m_i}]$ mapping each column c_i to an attribute a_i . An *attribute* $a_i = \ell_i(V_i)$ consists of a *level* ℓ_i and a *value* V_i , 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_i is actually an array of values, with one value per row. Column names c 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_i 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:

$\delta_{in} ::= [t_i \mapsto \tau_i \mid i \in 1..n]$	table map
$c, o ::= b_1.(\dots).b_n$	column name
$\tau ::= [c_i \mapsto a_i \mid i \in 1..m]$	table in database
$\rho_{sz} ::= [t_i \mapsto sz_i \mid i \in 1..n]$	table size map
$a ::= \ell(V)$	attribute value: V with level ℓ
$V ::= ? \mid s \mid [V_0, \dots, V_{n-1}]$	nullable value
$\ell, pc ::= \mathbf{static} \mid \mathbf{inst}$	level ($\mathbf{static} < \mathbf{inst}$)

4.2.2 Syntax of Core Schemas

We begin by giving the syntax of Core schemas, which have a straightforward interpretation as factor graphs and a direct semantics (presented later in this chapter). We first define the basic building blocks of a Tabular column.

Index Expressions, Spaces and Dependent Types of Tabular:

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

An *indexed 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, **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 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 $\mathbf{mod}(e)$, 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 indexed expression defining the size of the array. We use $\mathbf{link}(t)$ as a shorthand for $\mathbf{mod}(\mathbf{sizeof}(t))$.

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.

Expressions of Tabular:

$E, F ::=$	expression
e	index expression
$g(E_1, \dots, E_n)$	deterministic primitive g
$D[e_1, \dots, e_m](F_1, \dots, F_n)$	random draw from distribution D
if E then F_1 else F_2	if-then-else
$[E_1, \dots, E_n] \mid E[F]$	array literal, lookup
for $x < e \rightarrow F$	for loop (scope of index x is F)
infer . $D[e_1, \dots, e_m].c(E)$	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

The grammar of expressions, defining models of the particular columns of the table, is mostly standard for a first-order functional language. The expression $D[e_1, \dots, e_m](F_1, \dots, F_n)$ represents a random draw from a primitive distribution D with hyperparameters determined by the indexed expressions e_1, \dots, e_m and parameters defined by the expressions F_1, \dots, F_n . The operator **infer**. $D[e_1, \dots, e_m].c(E)$ returns an approximate value of the parameter c of the posterior distribution of expression E , expected to be of the form $D[e_1, \dots, e_m]$. 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 deterministic and random primitives. The deterministic primitives include the following:

Deterministic Primitives: $g : (x_1 : T_1, \dots, x_n : T_n) \rightarrow T$

$(>) : (x_1 : \mathbf{real!det}, x_2 : \mathbf{real!det}) \rightarrow \mathbf{bool!det}$
 $(>) : (x_1 : \mathbf{int!det}, x_2 : \mathbf{int!det}) \rightarrow \mathbf{bool!det}$
 $(=) : (x_1 : \mathbf{int!det}, x_2 : \mathbf{int!det}) \rightarrow \mathbf{bool!det}$
 $\mathbf{or} : (x_1 : \mathbf{bool!det}, x_2 : \mathbf{bool!det},) \rightarrow \mathbf{bool!det}$
 $(-) : (x_1 : \mathbf{real!det}, x_2 : \mathbf{real!det}) \rightarrow \mathbf{real!det}$
 $(-) : (x_1 : \mathbf{int!det}, x_2 : \mathbf{int!det}) \rightarrow \mathbf{int!det}$

Distribution signatures are parametrized by *spc*, to distinguish the use of corresponding distributions in random models and inside queries. This distinction was made to simplify typing rules for Tabular (shown in Section 4.4). The signatures of distributions include the following:

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

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

The names of parameters of distributions are fixed and not α -convertible, as they can be referenced by name by the **infer** operator. The lists of deterministic and random functions can be extended with any other operators and distributions. Moreover, we can include multiple signatures for different parametrisations of the same distribution—for instance, the Gaussian 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: $D_{spc} : [x_1 : T_1, \dots, x_m : T_m](c_1 : U_1, \dots, c_n : U_n) \rightarrow T$

The syntax of Core Tabular schemas is as follows:

Core Tabular Schemas:

$\mathbb{S} ::= [] \mid (t_1 = \mathbb{T}_1) :: \mathbb{S}$	(database) schema
$\mathbb{T} ::= [] \mid (c \triangleright x : T \ell \text{ viz } M) :: \mathbb{T}$	table (or function) (scope of x is \mathbb{T})
$\text{viz} ::= \text{input} \mid \text{local} \mid \text{output}$	visibility
$M, N ::= \varepsilon \mid E$	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 , an *internal name* x , 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 4.4), 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**.

In the rest of this chapter, col denotes a single column $(c \triangleright x : T \ell \text{ viz } M)$ 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 on which this chapter is based [Gordon et al., 2015], each column only had one name. This caused 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 (discussed in Section 4.3) 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 (with the latter possibly updated by substitution).

4.2.3 Syntax of Schemas with Functions and Indexing

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

A function is represented as a Core table whose last “return” 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 4.3) 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 “caller” column and c is the field name of the referenced column of the table (we need to use the field name, because the local name is only in scope in the function itself).

An *indexed expression* $M[e_{index} < e_{size}]$ 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:

$E ::= \dots \mid e.c$	expression
$M, N ::= \dots \mid M[e_{index} < e_{size}] \mid \mathbb{T} R$	model expression
$R ::= (c_1 = e_1, \dots, c_n = e_n)$	function arguments

The function field reference is only defined to be $e.c$ rather than $x.c$ in order for substitution to be well-defined (as described in Section 4.3.1.1). Note that 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.

4.2.3.1 Free Variables and Core Columns

The free variables in a table \mathbb{T} are all local variables used in model expressions which are not bound by column declarations or for-loops. They are formally defined as follows:

Free Variables: $\text{fv}(R)$, $\text{fv}(E)$, $\text{fv}(M)$, $\text{fv}(\mathbb{T})$:

$$\begin{aligned}\text{fv}(\[]) &= \emptyset \\ \text{fv}((c = E) :: R) &= \text{fv}(E) \cup \text{fv}(R) \\ \text{fv}(s) &= \emptyset \\ \text{fv}(x) &= x \\ \text{fv}(\text{sizeof}(t)) &= \emptyset \\ \text{fv}(g(E_1, \dots, E_n)) &= \text{fv}(E_1) \cup \dots \cup \text{fv}(E_n) \\ \text{fv}(D[e_1, \dots, e_m](F_1, \dots, F_n)) &= \text{fv}(e_1) \cup \dots \cup \text{fv}(e_m) \cup \text{fv}(F_1) \cup \dots \cup \text{fv}(F_n) \\ \text{fv}(\text{if } E \text{ then } F_1 \text{ else } F_2) &= \text{fv}(E) \cup \text{fv}(F_1) \cup \text{fv}(F_2) \\ \text{fv}([E_1, \dots, E_n]) &= \text{fv}(E_1) \cup \dots \cup \text{fv}(E_n) \\ \text{fv}(E[F]) &= \text{fv}(E) \cup \text{fv}(F) \\ \text{fv}([\text{for } x < e \rightarrow F]) &= \text{fv}(F) \setminus \{x\} \\ \text{fv}(\text{infer}.D[e_1, \dots, e_m].c(E)) &= \text{fv}(e_1) \cup \dots \cup \text{fv}(e_m) \cup \text{fv}(E) \\ \text{fv}(E : t.c) &= \text{fv}(E) \\ \text{fv}(t.c) &= \emptyset \\ \text{fv}(x.c) &= \{x\} \\ \text{fv}(\varepsilon) &= \emptyset \\ \text{fv}(\mathbb{T} R) &= \text{fv}(\mathbb{T}) \cup \text{fv}(R) \\ \text{fv}(M[e_{\text{index}} < e_{\text{size}}]) &= \text{fv}(M) \cup \text{fv}(e_{\text{index}}) \cup \text{fv}(e_{\text{size}}) \\ \text{fv}(\[]) &= \emptyset \\ \text{fv}((c \triangleright x : T \ell \text{ viz } M) :: \mathbb{T}) &= \text{fv}(T) \cup \text{fv}(M) \cup (\text{fv}(\mathbb{T}) \setminus \{x\}) \\ \text{fv}(\[]) &= \emptyset \\ \text{fv}((t = \mathbb{T}) :: \mathbb{S}) &= \text{fv}(\mathbb{T}) \cup \text{fv}(\mathbb{S})\end{aligned}$$

Note that unbound occurrences of field names are not considered as 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. Formally, we can define this operator by the following rules:

Core Attributes, Tables, and Schemas:

$$\begin{aligned}\text{Core}((c \triangleright x : T \ell \text{ input } \varepsilon)) \quad \text{Core}((c \triangleright x : T \ell \text{ local } E)) \quad \text{Core}((c \triangleright x : T \ell \text{ output } E)) \\ \text{Core}(\[]) \\ \text{Core}(\text{col} :: \mathbb{T}) \text{ if } \text{Core}(\text{col}) \text{ and } \text{Core}(\mathbb{T})\end{aligned}$$

$\text{Core}([])$
 $\text{Core}((t = \mathbb{T}) :: \mathbb{S}) \text{ if } \text{Core}(\mathbb{T}) \text{ and } \text{Core}(\mathbb{S})$

4.3 Reduction to Core Tabular

We now define the reduction relation reducing arbitrary well-typed Tabular schemas (with function applications and indexing) to a Core form.

Judgments:

$\mathbb{S} \rightarrow \mathbb{S}'$	schema reduction
$\mathbb{T} \rightarrow \mathbb{T}'$	table reduction
$M \rightarrow M'$	model reduction

As usual, we present a basic example before discussing the technical details. However, 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 very 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	GaussianFromMeanAndPrecision(M,P)
Prec ▷ Prec	real!rnd	static output	Gamma(1.0, 1.0)
ret ▷ ret	real!rnd	output	GaussianFromMeanAndPrecision(Mean, Prec)

Suppose we want to use this function to model eruptions of the Old Faithful geyser. The eruptions of this geyser, known for its regularity, can be split into two clusters based on their duration and waiting time: 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]

4.3.1 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 a 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 (referencing columns preceding the function application) 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	GaussianFromMeanAndPrecision(0.0,1.0)
duration.Prec ▷ Prec	real!rnd	static output	Gamma(1.0,1.0)
duration ▷ duration	real!rnd	output	GaussianFromMeanAndPrecision(Mean,Prec)
time.Mean ▷ Mean	real!rnd	static output	GaussianFromMeanAndPrecision(60.0,1.0)
time.Prec ▷ Prec	real!rnd	static output	Gamma(1.0,1.0)
time ▷ time	real!rnd	output	GaussianFromMeanAndPrecision(Mean,Prec)

Note that, 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`.

4.3.1.1 Binders and Capture-avoiding Substitution

In order to define the reduction rules formally, 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 .

The substitution $\mathbb{T}\{e/x\}$, together with auxiliary operators $E\{e/x\}$, $T\{e/x\}$, $R\{e/x\}$ and $M\{e/x\}$, is defined as follows:

Substitution: $E\{e/x\}, T\{e/x\}, R\{e/x\}, M\{e/x\}, \mathbb{T}\{e/x\}$

$$\begin{aligned}
y\{e/x\} &\triangleq \begin{cases} e & \text{if } y = x \\ y & \text{otherwise} \end{cases} \\
s\{e/x\} &\triangleq s \\
\mathbf{sizeof}(t)\{e/x\} &\triangleq \mathbf{sizeof}(t) \\
g(E_1, \dots, E_n)\{e/x\} &\triangleq g(E_1\{e/x\}, \dots, E_n\{e/x\}) \\
D[e_1, \dots, e_m](F_1, \dots, F_n)\{e/x\} &\triangleq D[e_1\{e/x\}, \dots, e_m\{e/x\}](F_1\{e/x\}, \dots, F_n\{e/x\}) \\
(\mathbf{if } E \mathbf{ then } F_1 \mathbf{ else } F_2)\{e/x\} &\triangleq \mathbf{if } E\{e/x\} \mathbf{ then } F_1\{e/x\} \mathbf{ else } F_2\{e/x\} \\
[E_1, \dots, E_n]\{e/x\} &\triangleq [E_1\{e/x\}, \dots, E_n\{e/x\}] \\
(E[F])\{e/x\} &\triangleq E\{e/x\}[F\{e/x\}] \\
[\mathbf{for } y < e' \rightarrow F]\{e/x\} &\triangleq \begin{cases} [\mathbf{for } y < e'\{e/x\} \rightarrow F] & \text{if } y = x \\ [\mathbf{for } y < e'\{e/x\} \rightarrow F\{e/x\}] & \text{otherwise if } y \notin \text{fv}(e) \end{cases} \\
\mathbf{infer}.D[e_1, \dots, e_m].c(E)\{e/x\} &\triangleq \mathbf{infer}.D[e_1\{e/x\}, \dots, e_m\{e/x\}].c(E\{e/x\}) \\
E : t.c\{e/x\} &\triangleq E[A, e] : t.c \\
t.c\{e/x\} &\triangleq t.c \\
e'.c\{e/x\} &\triangleq (e'\{e/x\}).c \\
S!spc\{e/x\} &\triangleq S!spc \\
T[e']\{e/x\} &\triangleq (T\{e/x\})[e'\{e/x\}] \\
\mathbf{mod}(e')\{e/x\} &\triangleq \mathbf{mod}(e'\{e/x\}) \\
[]\{e/x\} &\triangleq [] \\
((c = e') :: R)\{e/x\} &\triangleq (c = e'\{e/x\}) :: R\{e/x\} \\
\varepsilon\{e/x\} &\triangleq \varepsilon \\
M[e_1 < e_2]\{e/x\} &\triangleq M\{e/x\}[e_1\{e/x\} < e_2\{e/x\}] \\
(\mathbb{T} R)\{e/x\} &\triangleq (\mathbb{T}\{e/x\})(R\{e/x\}) \\
[]\{e/x\} &\triangleq []
\end{aligned}$$

$$\begin{aligned}
& ((c \triangleright y : T \ell \text{ viz } M) :: \mathbb{T}) \{e/x\} \triangleq \\
& \begin{cases} (c \triangleright y : T \{e/x\} \ell \text{ viz } M \{e/x\}) :: \mathbb{T} & \text{if } y = x \\ (c \triangleright y : T \{e/x\} \ell \text{ viz } M \{e/x\}) :: (\mathbb{T} \{e/x\}) & \text{if } y \neq x \text{ and } y \notin \text{fv}(e) \end{cases}
\end{aligned}$$

In the reduction of function applications, only variables referencing Core columns are ever substituted, so the case $e'.c \{e/x\}$ is only defined for mathematical completeness.

As usual, the substitution operator is applied to α -equivalence classes of terms, rather than ground terms themselves, so we can always assume that binders are not free in the substituted expression e . The substitution in α -equivalence classes of Tabular programs could be defined more formally using the theory of nominal sets, developed by Pitts [2006]. To this end, we would have to show that the sets of all Tabular expressions, types and tables are nominal sets (that is, sets whose all elements are supported² by some finite sets of atomic names) and define for each data constructor K a non-recursive function f_K taking separate components of an expression and returning the constructed expression with one step of the substitution (without recursing into subexpressions) performed. Then, if we showed that all functions f_K were supported by some single set A and did not introduce new fresh variables (which would, in fact, hold by inspection), by Theorem 5.1 from [Pitts, 2006] there would be a single family of functions \hat{f}_s (one for each data sort s) such that for every s , the function \hat{f}_s would unfold into repetitive applications of functions f_K to the subterms of the original term. This family would in fact define our substitution function.

A rigorous account of α -structural recursion for Tabular programs would, however, be very tedious and rather uninteresting, so it is left outside the scope of this dissertation. To simplify presentation, in the remainder of this chapter we will resort to the usual, slightly informal recursive definitions, which implicitly assume that all binders are fresh.

The substitution $\mathbb{T}\langle y/x.c \rangle$ of function field accesses in tables, together with the auxiliary operators, is defined below.

Function field access substitution: $E\langle y/x.c \rangle, M\langle y/x.c \rangle, \mathbb{T}\langle y/x.c \rangle$

$$z\langle y/x.c \rangle \triangleq z$$

²In this context, an element X of some nominal set is *supported* by A if for all $a, a' \notin A$, swapping a and a' in X has no effect

$$\begin{aligned}
s\langle y/x.c \rangle &\triangleq s \\
\mathbf{sizeof}(t)\langle y/x.c \rangle &\triangleq \mathbf{sizeof}(t) \\
g(E_1, \dots, E_n)\langle y/x.c \rangle &\triangleq g(E_1\langle y/x.c \rangle, \dots, E_n\langle y/x.c \rangle) \\
D[e_1, \dots, e_m](F_1, \dots, F_n) \{e/x\} &\triangleq D[e_1, \dots, e_m](F_1\langle y/x.c \rangle, \dots, F_n\langle y/x.c \rangle) \\
(\mathbf{if } E \mathbf{ then } F_1 \mathbf{ else } F_2)\langle y/x.c \rangle &\triangleq \mathbf{if } E\langle y/x.c \rangle \mathbf{ then } F_1\langle y/x.c \rangle \mathbf{ else } F_2\langle y/x.c \rangle \\
[E_1, \dots, E_n]\langle y/x.c \rangle &\triangleq [E_1\langle y/x.c \rangle, \dots, E_n\langle y/x.c \rangle] \\
(E[F])\langle y/x.c \rangle &\triangleq E\langle y/x.c \rangle[F\langle y/x.c \rangle] \\
[\mathbf{for } z < e' \rightarrow F]\langle y/x.c \rangle &\triangleq [\mathbf{for } z < e' \rightarrow F\langle y/x.c \rangle] \\
\mathbf{infer}.D[e_1, \dots, e_m].d(E)\langle y/x.c \rangle &\triangleq \mathbf{infer}.D[e_1, \dots, e_m].d(E\langle y/x.c \rangle) \\
E : t.d\langle y/x.c \rangle &\triangleq E\langle z/x.c \rangle : t.d \\
t.d\langle y/x.c \rangle &\triangleq t.d \\
e.d\langle y/x.c \rangle &\triangleq \begin{cases} y & \text{if } e = x \text{ and } d = c \\ e.d & \text{otherwise} \end{cases} \\
\mathcal{E}\langle y/x.c \rangle &\triangleq \mathcal{E} \\
M[e_1 < e_2]\langle y/x.c \rangle &\triangleq M\langle y/x.c \rangle[e_1 < e_2] \\
(\mathbb{T} R)\langle y/x.c \rangle &\triangleq (\mathbb{T}\langle y/x.c \rangle) R \\
((c \triangleright z : T \ell \text{ viz } M) :: \mathbb{T})\langle y/x.c \rangle &\triangleq \\
\begin{cases} (c \triangleright z : T \ell \text{ viz } M\langle y/x.c \rangle) :: \mathbb{T} & \text{if } y = x \\ (c \triangleright z : T \ell \text{ viz } M\langle y/x.c \rangle) :: (\mathbb{T}\langle y/x.c \rangle) & \text{otherwise if } y \notin \text{fv}(e) \end{cases} \\
[]\langle y/x.c \rangle &\triangleq []
\end{aligned}$$

Note that we do not need to define field access substitutions in argument lists and types, because they cannot contain any expressions of the form $x.c$ by the definition of the syntax.

To illustrate how field reference substitution works, consider again the simplified version of the Old Faithful model from the beginning of Section 4.3.1, but this time using different local variable and field names, to emphasise the fact that they are not the same thing:

table Faithful			
duration \triangleright x	real!rnd	output	CG(M=0.0, P=1.0)
time \triangleright y	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 4.1.3). 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 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 in the next section), 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) column `x.Mean` in the rest of the table with the variable y 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.

4.3.1.2 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. In the reduction rules, we normally use o for the name of the “caller” column and c for the name of a column in the function table, to disambiguate between the two.

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

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

$y \notin \text{fv}(\mathbb{T}', R) \cup \{x\} \quad c \neq \text{ret}$

$$\frac{(o \triangleright x : T \ell \text{ viz } ((c \triangleright y : T' \ell' \text{ output } E) :: \mathbb{T}) R) :: \mathbb{T}' \rightarrow}{(o.c \triangleright y : T' (\ell \wedge \ell') \text{ viz } E) :: (o \triangleright x : T \ell \text{ viz } \mathbb{T} R) :: \mathbb{T}' \langle y/x.c \rangle}$$

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

$y \notin \text{fv}(\mathbb{T}', R) \cup \{x\}$

$$\frac{(o \triangleright x : T \ell \text{ viz } ((c \triangleright y : T' \ell' \text{ local } E) :: \mathbb{T}) R) :: \mathbb{T}' \rightarrow}{(_ \triangleright y : T' (\ell \wedge \ell') \text{ local } E) :: (o \triangleright x : T \ell \text{ viz } \mathbb{T} R) :: \mathbb{T}'}$$

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

$$\frac{(o \triangleright x : T \ell \text{ viz } (c \triangleright y : T' \ell' \text{ input } \varepsilon) :: \mathbb{T} (c = e) :: R) :: \mathbb{T}' \rightarrow}{(o \triangleright x : T \ell \text{ viz } \mathbb{T} \{e/y\} R) :: \mathbb{T}'}$$

(RED APPL RET)

$$\frac{(o \triangleright x : T \ell \text{ viz } [(\text{ret} \triangleright y : T' \ell' \text{ output } E)] []) :: \mathbb{T}' \rightarrow}{(o \triangleright x : T' (\ell \wedge \ell') \text{ viz } E) :: \mathbb{T}'}$$

(RED TABLE RIGHT)

$\mathbb{T} \rightarrow \mathbb{T}' \quad \text{Core}(\text{col})$

$\text{col} :: \mathbb{T} \rightarrow \text{col} :: \mathbb{T}'$

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 next step). 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 table, y is required to be fresh in \mathbb{T}' and R . This is always possible, because

tables are identified up to alpha-conversion of internal column names, so y can be refreshed if needed (formally speaking, 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 version of the Old Faithful example used in Section 4.3.1.1, 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	GaussianFromMeanAndPrecision(0.0, 1.0)
Prec ▷ Prec	real!rnd	static output	Gamma(1.0, 1.0)
ret ▷ ret	real!rnd	output	GaussianFromMeanAndPrecision(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:

table Faithful			
duration.Mean ▷ Mean	real!rnd	static output	GaussianFromMeanAndPrecision(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	GaussianFromMeanAndPrecision(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.

4.3.2 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 of 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 → GaussianFromMeanAndPrecision(M,P)]
Prec ▷ Prec	real!rnd	static output	[for _ < 2 → Gamma(1.0, 1.0)]
ret ▷ ret	real!rnd	output	GaussianFromMeanAndPrecision(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 \triangleright Mean	real!rnd[2]	static output	[for _ < 2 \rightarrow GaussianFromMeanAndPrecision(0.0,1.0)]
duration.Prec \triangleright Prec	real!rnd[2]	static output	[for _ < 2 \rightarrow Gamma(1.0,1.0)]
duration \triangleright duration	real!rnd	output	GaussianFromMeanAndPrecision (Mean[cluster], Prec[cluster])

More generally, table indexing is formalized 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. The predicate NoQry states that a given Core table or model has no **qry**-level columns.

Tables without query columns: $\text{NoQry}(\mathbb{T})$, $\text{NoQry}(M)$ for $M \neq E$

$\text{NoQry}([])$
 $\text{NoQry}((c \triangleright x : T \ell \text{ viz } E) :: \mathbb{T}) \text{ iff } \neg \text{qry}(T) \text{ and } \text{NoQry}(\mathbb{T})$
 $\text{NoQry}(\mathbb{T} R) \text{ iff } \text{NoQry}(\mathbb{T})$
 $\text{NoQry}(M[e_1 < e_2]) \text{ iff } \text{NoQry}(M)$

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), occurring outside parts of syntax where an indexed expression e or a variable is expected, replaced with the array access $x[e]$.

More formally, this operator is defined as follows:

Expression indexing: $E[A, e]$

$$x[A, e] \triangleq \begin{cases} x[e] & \text{if } x \in A \\ x & \text{otherwise} \end{cases}$$

$$c[A, e] \triangleq c$$

$$\text{sizeof}(t)[A, e] \triangleq \text{sizeof}(t)$$

$$g(E_1, \dots, E_n)[A, e] \triangleq g(E_1[A, e], \dots, E_n[A, e])$$

$$\begin{aligned}
D[e_1, \dots, e_m](F_1, \dots, F_n)[A, e] &\triangleq D[e_1, \dots, e_m](F_1[A, e], \dots, F_n[A, e]) \\
(\text{if } E \text{ then } F_1 \text{ else } F_2)[A, e] &\triangleq \text{if } E[A, e] \text{ then } F_1[A, e] \text{ else } F_2[A, e] \\
[E_1, \dots, E_n][A, e] &\triangleq [E_1[A, e], \dots, E_n[A, e]] \\
(E[F])[A, e] &\triangleq E[A, e][F[A, e]] \\
[\text{for } x < e' \rightarrow F][A, e] &\triangleq [\text{for } x < e' \rightarrow F[A, e]] \quad \text{if } x \notin \text{fv}(e) \cup A \\
\text{infer}.D[e_1, \dots, e_m].y(E)[A, e] &\triangleq \text{infer}.D[e_1, \dots, e_m].y(E[A, e]) \\
E : t.c[A, e] &\triangleq E[A, e] : t.c \\
t.c[A, e] &\triangleq t.c \\
x.c[A, e] &\triangleq x.c
\end{aligned}$$

We do not need to worry about variables which cannot be replaced with expressions other than index expressions due to syntax restrictions, 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.

The case $\text{infer}.D[e_1, \dots, e_m].y(E)[A, e]$ is only defined for mathematical completeness, as the **infer** operator can only be used in **qry**-level columns, which cannot be indexed by assumption.

The indexing operator is defined inductively below.

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

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

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 other **static rnd** columns of the original table,

which have been turned into arrays, we must replace these references (by means of the $[\cdot]$ 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 remaining reduction rules required to reduce indexed expressions:

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

<p>(RED INDEX)</p> $\frac{\text{Core}(\mathbb{T}) \quad \text{NoQry}(\mathbb{T})}{(\mathbb{T} \ R)[e_{\text{index}} < e_{\text{size}}] \rightarrow (\text{index}_{\emptyset}(\mathbb{T}, e_{\text{index}}, e_{\text{size}})) \ R}$	<p>(RED INDEX INNER)</p> $\frac{M \rightarrow M'}{M[e_{\text{index}} < e_{\text{size}}] \rightarrow M'[e_{\text{index}} < e_{\text{size}}]}$
<p>(RED INDEX EXPR)</p> $\frac{}{E[e_{\text{index}} < e_{\text{size}}] \rightarrow E}$	

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

<p>(RED MODEL)</p> $\frac{M \rightarrow M'}{(c \triangleright x : T \ \ell \ \text{viz} \ M) :: \mathbb{T} \rightarrow (c \triangleright x : T \ \ell \ \text{viz} \ M') :: \mathbb{T}}$
--

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 basic expressions have no static parameters of their own, indexing a basic expressions 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.

4.3.3 Reducing Schemas

Finally, we have two reduction rules for schemas:

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

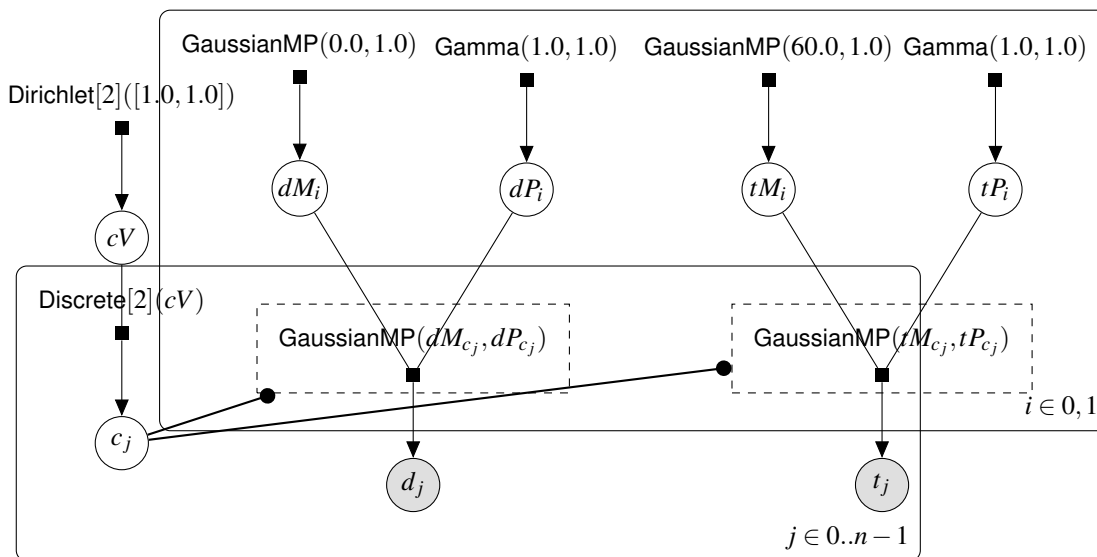
(RED SCHEMA LEFT)	(RED SCHEMA RIGHT)
$\frac{\mathbb{T} \rightarrow \mathbb{T}'}{(t = \mathbb{T}) :: \mathbb{S} \rightarrow (t = \mathbb{T}') :: \mathbb{S}}$	$\frac{\mathbb{S} \rightarrow \mathbb{S}' \quad \text{Core}(\mathbb{T})}{(t = \mathbb{T}) :: \mathbb{S} \rightarrow (t = \mathbb{T}) :: \mathbb{S}'}$

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 \triangleright V	real!rnd[2]	static output	Dirichlet[2]([for i < 2 \rightarrow 1.0])
cluster \triangleright cluster	mod(2)!rnd	output	Discrete[2](V)
duration.Mean \triangleright Mean	real!rnd[2]	static output	[for i < 2 \rightarrow GaussianFromMeanAndPrecision(0.0, 1.0)]
duration.Prec \triangleright Prec	real!rnd[2]	static output	[for i < 2 \rightarrow Gamma(1.0, 1.0)]
duration \triangleright duration	real!rnd	output	GaussianFromMeanAndPrecision(Mean[cluster], Prec[cluster])
time.Mean \triangleright Mean	real!rnd[2]	static output	[for i < 2 \rightarrow GaussianFromMeanAndPrecision(60.0, 1.0)]
time.Prec \triangleright Prec	real!rnd[2]	static output	[for i < 2 \rightarrow Gamma(1.0, 1.0)]
time \triangleright time	real!rnd	output	GaussianFromMeanAndPrecision(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 following (directed) factor graph, 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 $\text{GaussianMP}(0.0, 1.0)$. The boxes with dotted lines are *gates* [Minka and Winn, 2008], 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.

4.4 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.

In this section, we define the Tabular type system formally and present the type soundness property of the reduction system shown in Section 4.3 (deferring the detailed proof until Appendix B).

In addition to the column types introduced in Section 4.2, 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 runtime, in which case an error message from the inference backend is shown to the user in the implementation.

4.4.1 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 global), column types, levels and visibilities. 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. Note that these types are notably simpler than the nested record types used in the original formulation of Tabular [Gordon et al., 2014].

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:

$Q ::= [] \mid (c \triangleright x : T \ell \text{ viz}) :: Q$	table type (scope of x is Q , $\text{viz} \neq \mathbf{local}$)
$Sty ::= (t : Q) :: Sty$	schema type
fun (Q) iff $\text{viz}_n = \mathbf{output}$ and $c_n = \mathbf{ret}$	
table (Q) iff for each $i \in 1..n$, $\ell_i = \mathbf{static} \Rightarrow \mathbf{rnd}(T_i) \vee \mathbf{qry}(T_i)$	
red (Q) iff table (Q) and for each $i \in 1..n$, $\text{viz}_i = \mathbf{output}$	

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 extend the definition of fv to Q -types:

Free Variables: $\text{fv}(Q)$

$\text{fv}([]) = \emptyset$
$\text{fv}((c \triangleright x : T \ell \text{ viz}) :: Q) = \text{fv}(T) \cup (\text{fv}(Q) \setminus \{x\})$

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

Tabular Typing Environments:

$\Gamma ::= \emptyset \mid (\Gamma, x :^\ell T) \mid (\Gamma, t : Q) \mid (\Gamma, x : Q)$	environment
--	-------------

The *domain* $\text{dom}(\Gamma)$ of an environment Γ is the set of all variable and table names in the environment:

Domain of an Environment:

$\text{dom}(\emptyset) = \emptyset$
$\text{dom}(\Gamma, x :^\ell T) = \{x\} \cup \text{dom}(\Gamma)$
$\text{dom}(\Gamma, t : Q) = \{t\} \cup \text{dom}(\Gamma)$
$\text{dom}(\Gamma, x : Q) = \{x\} \cup \text{dom}(\Gamma)$

Below is the list of all judgments of the Tabular type systems, which will be described in detail 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
$\Gamma \vdash^{pc} R : Q \rightarrow Q'$	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. A formal definition of α -equivalence in Tabular can be found in Appendix A.

4.4.2 Type Well-formedness and Expression Types

We begin with the well-formedness rules for environments and column types and typing rules for indexed expressions (which are mutually dependent on each other). Below, $\text{ty}(s)$ denotes the scalar type of the scalar s : **real** if s is a real number, **int** if it is an integer and **bool** if it is a Boolean. The symbol @ denotes concatenation of Q -types

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

(ENV EMPTY)	(ENV VAR)	(ENV FUN) ($\text{red}(Q)$)	(ENV TABLE) ($\text{table}(Q)$)
$\frac{}{\emptyset \vdash \diamond}$	$\frac{\Gamma \vdash T \quad x \notin \text{dom}(\Gamma)}{\Gamma, x :^{pc} T \vdash \diamond}$	$\frac{\Gamma \vdash Q \quad x \notin \text{dom}(\Gamma)}{\Gamma, x : Q \vdash \diamond}$	$\frac{\Gamma \vdash Q \quad t \notin \text{dom}(\Gamma)}{\Gamma, t : Q \vdash \diamond}$
(TYPE SCALAR)	(TYPE RANGE)	(TYPE ARRAY)	
$\frac{\Gamma \vdash \diamond}{\Gamma \vdash S! \text{spc}}$	$\frac{\Gamma \vdash^{\text{static}} e : \text{int}! \text{det}}{\Gamma \vdash \text{mod}(e)! \text{spc}}$	$\frac{\Gamma \vdash T \quad \Gamma \vdash^{\text{static}} e : \text{int}! \text{det}}{\Gamma \vdash T[e]}$	
(INDEX VAR) (for $\ell \leq pc$)	(INDEX SCALAR)	(INDEX MOD)	
$\frac{\Gamma \vdash \diamond \quad \Gamma = \Gamma_1, x :^\ell T, \Gamma_2}{\Gamma \vdash^{pc} x : T}$	$\frac{\Gamma \vdash \diamond \quad S = \text{ty}(s)}{\Gamma \vdash^{pc} s : S! \text{det}}$	$\frac{\Gamma \vdash \diamond \quad 0 \leq n < m}{\Gamma \vdash^{pc} n : \text{mod}(m)! \text{det}}$	
(INDEX SIZEOF)	(FUNREFRET)		
$\frac{\Gamma \vdash \diamond \quad \Gamma = \Gamma', t : Q, \Gamma''}{\Gamma \vdash^{pc} \text{sizeof}(t) : \text{int}! \text{det}}$	$\frac{\Gamma \vdash \diamond \quad \Gamma = \Gamma', x : Q, \Gamma'' \quad Q = Q' @ [(\text{ret} \triangleright y : T \ell \text{ output})] \quad \ell \leq pc}{\Gamma \vdash^{pc} x : T}$		

The (ENV EMPTY), (ENV VAR), (ENV FUN) and (ENV TABLE) rules state that an environment is well typed if and only if its variables are unique, all column and table types are well-formed (in the preceding part of the environment), all table types satisfy the **table** predicate and all reduced function types satisfy **red**. The (TYPE SCALAR) rule says that a scalar type is well-formed in any well-formed environment, while (TYPE RANGE) and (TYPE ARRAY) state that only **static**, deterministic, integer-valued index expressions can appear in types. The (INDEX VAR) rule imposes the restriction that only **static**-level variables can be used in **static** expressions.

The rules (INDEX SCALAR) and (INDEX MOD) are straightforward, while (INDEX SIZEOF) states that table sizes are treated as integers. Finally, (FUNREFRET) allows to access the return column of a application via the local name of the “caller” column.

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 \square)	(TABLE TYPE INPUT)	(TABLE TYPE OUTPUT)
$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \square}$	$\frac{\Gamma \vdash T \quad \Gamma, x :^\ell T \vdash Q \quad c \notin \text{names}(Q)}{\Gamma \vdash (c \triangleright x : T \ell \text{ input}) :: Q}$	$\frac{\Gamma \vdash T \quad \Gamma, x :^\ell T \vdash Q \quad c \notin \text{names}(Q)}{\Gamma \vdash (c \triangleright x : T \ell \text{ output}) :: Q}$
(SCHEMA TYPE \square)	(SCHEMA TYPE TABLE)	
$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \square}$	$\frac{\Gamma \vdash Q \quad \text{table}(Q) \quad \Gamma, t : Q \vdash Sty}{\Gamma \vdash (t : Q) :: Sty}$	

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**, but not the other way round, nor between **rnd** and **qry**. 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 **det** \leq **rnd** and **det** \leq **qry**. We also define a (partial) least upper bound $spc \vee spc'$.

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

$spc \vee spc = spc \quad \text{det} \vee \text{rnd} = \text{rnd} \quad \text{det} \vee \text{qry} = \text{qry}$
(The combination rnd \vee qry is intentionally not defined.)

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

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

$$\begin{aligned} (S!spc) \vee spc' &\triangleq S!(spc \vee spc') & T[e] \vee spc &\triangleq (T \vee spc)[e] \\ (\mathbf{mod}(e)!spc) \vee spc' &\triangleq \mathbf{mod}(e)!(spc \vee spc') \end{aligned}$$

With these operations in place, we can define the subtyping rules:

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

(SUB SCALAR)	(SUB MOD)	(SUB ARRAY)
$\frac{\Gamma \vdash \diamond \quad spc_1 \leq spc_2}{\Gamma \vdash S!spc_1 <: S!spc_2}$	$\frac{\Gamma \vdash^{\mathbf{static}} e : \mathbf{int}! \mathbf{det} \quad spc_1 \leq spc_2}{\Gamma \vdash \mathbf{mod}(e)!spc_1 <: \mathbf{mod}(e)!spc_2}$	$\frac{\Gamma \vdash T <: U \quad \Gamma \vdash^{\mathbf{static}} e : \mathbf{int}! \mathbf{det}}{\Gamma \vdash T[e] <: U[e]}$

These rules state that $\Gamma \vdash T <: U$ if and only if both T and U are well-formed in Γ , they are of the same form and $\text{space}(T) \leq \text{space}(U)$. This implies that we can use a **det** value when a **rnd** or **qry** value is expected.

Below, we present the 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.

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

(SUBSUM)	(INDEX EXPRESSION)
$\frac{\Gamma \vdash^{pc} E : T \quad \Gamma \vdash T <: U}{\Gamma \vdash^{pc} E : U}$	$\frac{\Gamma \vdash^{pc} e : T \quad (e \text{ is an index expression})}{\Gamma \vdash^{pc} e : T \quad (e \text{ seen as an expression})}$
(DEREF STATIC) $\frac{\Gamma \vdash \diamond \quad \Gamma = \Gamma', t : Q, \Gamma'' \quad Q = Q' @ [(c \triangleright x : T \mathbf{static} \text{ viz})] @ Q''}{\Gamma \vdash^{pc} t.c : T}$	(DEREF INST) $\frac{\Gamma \vdash^{pc} E : \mathbf{link}(t)!spc \quad \Gamma = \Gamma', t : Q, \Gamma'' \quad Q = Q' @ [(c \triangleright x : T \mathbf{inst} \text{ viz})] @ Q''}{\Gamma \vdash^{pc} E : t.c : T \vee spc}$
(RANDOM) (where $\sigma(U) \triangleq U\{e_1/x_1\} \dots \{e_m/x_m\}$) $D_{\mathbf{rnd}} : [x_1 : T_1, \dots, x_m : T_m](c_1 : U_1, \dots, c_n : U_n) \rightarrow T$ $\frac{\Gamma \vdash^{\mathbf{static}} e_i : T_i \quad \forall i \in 1..m \quad \Gamma \vdash^{pc} F_j : \sigma(U_j) \quad \forall j \in 1..n \quad \Gamma \vdash \diamond \quad \{x_1, \dots, x_m\} \cap (\bigcup_i \text{fv}(e_i)) = \emptyset \quad x_i \neq x_j \text{ for } i \neq j}{\Gamma \vdash^{pc} D[e_1, \dots, e_m](F_1, \dots, F_n) : \sigma(T)}$	

(ITER) (where $x \notin \text{fv}(T)$)	(INDEX)
$\Gamma \vdash^{\text{static}} e : \text{int} ! \text{det}$	$\text{space}(T) \leq \text{spc}$
$\Gamma, x : ^{pc} (\text{mod}(e) ! \text{det}) \vdash^{pc} F : T$	$\Gamma \vdash^{pc} E : T[e] \quad \Gamma \vdash^{pc} F : \text{mod}(e) ! \text{spc}$
$\Gamma \vdash^{pc} [\text{for } x < e \rightarrow F] : T[e]$	$\Gamma \vdash^{pc} E[F] : T \vee \text{spc}$
(INFER) (where $\sigma(U) \triangleq U\{e_1/x_1\} \dots \{e_m/x_m\}$)	
$D_{\text{qry}} : [x_1 : T_1, \dots, x_m : T_m](c_1 : U_1, \dots, c_n : U_n) \rightarrow T$	
$\Gamma \vdash^{\text{static}} e_i : T_i \quad \forall i \in 1..m \quad \Gamma \vdash^{pc} E : \sigma(T) \quad j \in 1..n$	
$\{x_1, \dots, x_m\} \cap (\bigcup_i \text{fv}(e_i)) = \emptyset \quad x_i \neq x_j \text{ for } i \neq j$	
$\Gamma \vdash^{pc} \text{infer}.D[e_1, \dots, e_m].c_j(E) : \sigma(U_j)$	
(PRIM)	
$\Gamma \vdash \diamond \quad g : (x_1 : T_1, \dots, x_n : T_n) \rightarrow T \quad \Gamma \vdash^{pc} E_i : T_i \vee \text{spc} \quad \forall i \in 1..n$	
$x_i \neq x_j \text{ for } i \neq j$	
$\Gamma \vdash^{pc} g(E_1, \dots, E_n) : T \vee \text{spc}$	
(IF)	
$\Gamma \vdash^{pc} E_1 : (\text{bool} ! \text{spc}) \quad \Gamma \vdash^{pc} E_2 : T \quad \Gamma \vdash^{pc} E_3 : T \quad \text{space}(T) \leq \text{spc}$	
$\Gamma \vdash^{pc} \text{if } E_1 \text{ then } E_2 \text{ else } E_3 : T \vee \text{spc}$	
(ARRAY)	
$\Gamma \vdash \diamond \quad \Gamma \vdash^{pc} E_i : T \quad \forall i \in 0..n-1$	
$\Gamma \vdash^{pc} [E_0, \dots, E_{n-1}] : T[n]$	
(FUNREF)	
$\Gamma \vdash \diamond \quad \Gamma = \Gamma', x : Q, \Gamma''$	
$Q = Q' @ [(c \triangleright y : T \ell \text{ viz})] @ Q''$	
$\ell \leq pc \quad x \neq \text{ret}$	
$\Gamma \vdash^{pc} x.c : T$	

The (SUBSUM) rule is a standard subsumption rule, which, in conjunction with the subtyping rules, allows **det**-level data to be used where **rnd** or **qry**-level data is expected. The (INDEX EXPRESSION) rule simply says that every valid typing judgment for an index expression e is also a valid judgement at the level of expressions. 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 refer-

ence 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 (ARRAY), (ITER) and (INDEX) rules are standard, except that, like in (DEREF INST), we need to join T with spc , because an array access is deterministic only if both the array and the index are deterministic. Similarly, in (IF), the space of T depends on determinacy of both the guard and the branches. The (PRIM) rule allows applying deterministic primitives to **rnd** and **qry** variables (but not a combination of the two) and changes the space of the output accordingly in these cases (recall that all the argument and return types of deterministic functions are assumed to be in **det**-space).

The (RANDOM) rule requires all hyperparameters of a distribution to be static. Since the types of parameters and the output type may depend on them, we need to substitute the values of hyperparameters in these types.

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 usually referencing a previous column), matches the output type of the distribution D , and returns the type of argument c_j (with appropriate substitution performed). 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.

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 .

4.4.3 Model Types

Before we extend the type system to compound models, we define typing rules for function argument lists. The judgment $\Gamma \vdash^{pc} R : Q \rightarrow Q'$ 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: $\Gamma \vdash^{pc} R : Q \rightarrow Q'$

$$\text{(ARG INPUT)} \frac{\Gamma \vdash^{\ell \wedge pc} e : T \quad \Gamma \vdash^{pc} R : Q\{e/x\} \rightarrow Q'}{\Gamma \vdash^{pc} ((c = e) :: R) : ((c \triangleright x : T \ell \text{ input}) :: Q) \rightarrow Q'}$$

$$\begin{array}{c}
\text{(ARG OUTPUT)} \frac{\Gamma, x : {}^{\ell \wedge pc} T \vdash^{pc} R : Q \rightarrow Q' \quad c \neq \text{ret} \quad x \notin \text{fv}(R)}{\Gamma \vdash^{pc} R : ((c \triangleright x : T \ell \text{ output}) :: Q) \rightarrow ((c \triangleright x : T (\ell \wedge pc) \text{ output}) :: Q')} \\
\\
\text{(ARG RET)} \frac{\Gamma \vdash T}{\Gamma \vdash^{pc} R : (\text{ret} \triangleright x : T \ell \text{ output}) \rightarrow (\text{ret} \triangleright x : T (\ell \wedge pc) \text{ output})}
\end{array}$$

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 type ℓ is **static**, e must be static by definition, 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: $Q[e]$

$$\begin{array}{l}
\emptyset[e] \triangleq \emptyset \\
((c \triangleright x : T \text{ inst viz}) :: Q)[e] \triangleq (c \triangleright x : T \text{ inst viz}) :: (Q[e]) \quad \text{if } x \notin \text{fv}(e) \\
((c \triangleright x : T \text{ static viz}) :: Q)[e] \triangleq (c \triangleright x : T \text{ static viz}) :: (Q[e]) \\
\quad \text{if viz = input or det}(T) \text{ and } x \notin \text{fv}(e) \\
((c \triangleright x : T \text{ static viz}) :: Q)[e] \triangleq (c \triangleright x : T[e] \text{ static viz}) :: (Q[e]) \\
\quad \text{if viz} \neq \text{input} \text{ and } \text{rnd}(T) \text{ and } x \notin \text{fv}(e)
\end{array}$$

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

Table and Schema Types:

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

where @ denotes table concatenation.

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

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

(MODEL APPL)

$$\frac{\Gamma \vdash^{pc} \mathbb{T} : Q \quad \mathbf{fun}(\mathbb{T}) \quad \Gamma \vdash^{pc} R : Q \rightarrow Q'}{\Gamma \vdash^{pc} \mathbb{T} R : Q'}$$

(MODEL INDEXED)

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

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 4.3, only tables with no **qry** columns can be indexed, so we need to ensure that the table nested in M satisfies NoQry.

4.4.4 Table Types

The rules below are used for typechecking both top-level tables and function tables, which can be called from a **static** column, 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 $\ell \wedge pc$.

4.4.4.1 Tables with Core columns

We start with rules for typechecking Core columns.

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

(TABLE \square) (TABLE INPUT)

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash^{pc} \square : \square} \quad \frac{\Gamma, x : \ell \wedge pc \vdash^{pc} T \vdash^{pc} \mathbb{T} : Q \quad c \notin \mathbf{names}(Q)}{\Gamma \vdash^{pc} (c \triangleright x : T \ell \mathbf{input} \varepsilon) :: \mathbb{T} : (c \triangleright x : T (\ell \wedge pc) \mathbf{input}) :: Q}$$

(TABLE CORE OUTPUT)

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

(TABLE CORE LOCAL) (where $x \notin \text{fv}(Q)$)

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

The (TABLE \square) rule is obvious. The (TABLE INPUT) rule just adds the variable x to the environment (at level $\ell \wedge \text{viz}$) and checks the rest of the table.

The (TABLE CORE OUTPUT) rule checks the model expression E and then type-checks 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 4.1.2 and CGaussian from 4.3. 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 $\Gamma = N :^{\text{static}} \text{int!det}, R :^{\text{static}} \text{real!det}$. Then, by (ITER) and (RANDOM), we can show that

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

By applying (RANDOM) again, we get

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

By (TABLE CORE OUTPUT), the last column has type:

ret ▷ ret	mod(N)!rnd	output
-----------	------------	--------

in the environment $\Gamma, V :^{\text{static}} \text{real!rnd}[N]$. 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 4.1.2, 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

$$\emptyset \vdash^{\text{inst}} (N = 2, R = 1.0) : Q_{CD} \rightarrow Q'_{CD}$$

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} :

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

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

Mean \triangleright Mean	real!rnd	static output
Prec \triangleright Prec	real!rnd	static output
ret \triangleright ret	real!rnd	output

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

Mean \triangleright Mean	real!rnd[2]	static output
Prec \triangleright Prec	real!rnd[2]	static output
ret \triangleright ret	real!rnd	output

4.4.4.2 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: $c.Q$

$$\begin{aligned}
 c.((d \triangleright x : T \ell \text{ viz}) :: Q) &= (c.d \triangleright x : T \ell \text{ viz}) :: c.Q && \text{if } d \neq \text{ret} \\
 c.([\text{ret} \triangleright x : T \ell \text{ viz}]) &= [(c \triangleright x : T \ell \text{ viz})] \\
 c.([(d \triangleright x : T \ell \text{ viz})]) &= [(c.d \triangleright x : T \ell \text{ viz})] && \text{if } d \neq \text{ret}
 \end{aligned}$$

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

$$\begin{aligned}
 &(\text{TABLE OUTPUT}) \\
 &\Gamma \vdash^{\ell \wedge pc} M : Q_c \quad \Gamma, x : Q_c \vdash^{pc} \mathbb{T} : Q \quad Q_c = Q'_c @ [(\text{ret} \triangleright y : T \ell' \text{ output})] \\
 &\text{names}(c.Q_c) \cap \text{names}(Q) = \emptyset \\
 &\hline
 &\Gamma \vdash^{pc} (c \triangleright x : T \ell \text{ output } M) :: \mathbb{T} : (c.Q_c) @ Q \\
 &(\text{TABLE LOCAL}) \\
 &\Gamma \vdash^{\ell \wedge pc} M : Q_c \quad \Gamma, x : Q_c \vdash^{pc} \mathbb{T} : Q \quad Q_c = Q'_c @ [(\text{ret} \triangleright y : T \ell' \text{ output})] \\
 &\hline
 &\Gamma \vdash^{pc} (\varepsilon \triangleright x : T \ell \text{ local } M) :: \mathbb{T} : Q
 \end{aligned}$$

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 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 4.3.1 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

Example: accessing function fields Let us consider once again the version of the Old Faithful model from Section 4.3.1, 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 \mathcal{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

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

By (INFER), this only holds if

$$\Gamma \vdash^{\text{inst}} x.\text{Mean} : \text{real} ! \text{rnd}$$

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).

4.4.5 Schema Types

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

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

(SCHEMA \square)	(SCHEMA TABLE)
$\Gamma \vdash \diamond$	$\Gamma \vdash^{\text{inst}} \mathbb{T} : Q \quad \text{table}(Q) \quad \Gamma, t : Q \vdash \mathbb{S} : \text{Sty}$
$\Gamma \vdash \square : \square$	$\Gamma \vdash (t = \mathbb{T}) :: \mathbb{S} : (t : Q) :: \text{Sty}$

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.

4.4.6 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. As the proofs of these results are lengthy and require multiple auxiliary lemmas, they are omitted from the main part of this dissertation and can instead be found in Appendix B.

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 1 (Type preservation) (1) *If $\Gamma \vdash^{pc} M : Q$ and $M \rightarrow M'$, then $\Gamma \vdash^{pc} M' : Q$*

(2) *If $\Gamma \vdash^{inst} \mathbb{T} : Q$ and $\mathbb{T} \rightarrow \mathbb{T}'$, then $\Gamma \vdash^{inst} \mathbb{T}' : Q$*

(3) *If $\Gamma \vdash \mathbb{S} : Sty$ and $\mathbb{S} \rightarrow \mathbb{S}'$, then $\Gamma \vdash \mathbb{S}' : Sty$.*

Proof: In Appendix B. ■

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

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

(2) *If $\Gamma \vdash^{pc} \mathbb{S} : Sty$ then either $\text{Core}(\mathbb{S})$ or there is \mathbb{S}' such that $\mathbb{S} \rightarrow \mathbb{S}'$.*

Proof: In Appendix B. ■

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

Proposition 3 (Termination) *There does not exist an infinite chain of reductions $\mathbb{S}_1 \rightarrow \mathbb{S}_2 \rightarrow \dots$*

Proof: In Appendix B. ■

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

Theorem 1 *If $\emptyset \vdash \mathbb{S} : Sty$, then $\mathbb{S} \rightarrow^* \mathbb{S}'$ for some unique \mathbb{S}' such that $\text{Core}(\mathbb{S}')$ and $\emptyset \vdash \mathbb{S}' : Sty$.*

Proof: By Propositions 1 and 2, we can construct a maximal chain of reductions $\mathbb{S} \rightarrow \mathbb{S}_1 \rightarrow \mathbb{S}_2 \dots$ such that $\emptyset \vdash \mathbb{S}_i : Sty$ for all i and either $\text{Core}(\mathbb{S}_i)$ or $\mathbb{S}_i \rightarrow \mathbb{S}_{i+1}$. By Proposition 3, we know that this chain must be finite, so we must have $\text{Core}(\mathbb{S}_i)$ for some \mathbb{S}_i . The uniqueness of this \mathbb{S}_i follows from the determinacy of the reduction rules. ■

4.5 Semantics

In this section, we present the semantics of Core Tabular. Following the philosophy of Tabular, according to which a probabilistic program defines a query on the marginal distributions of random variables, the semantics is a database containing the most likely values of the requested parameters of marginals and the values of pseudo-deterministic expressions depending on these parameters.

The semantics is defined in two parts. We first introduce a *sampling semantics* computing the values of expressions in **infer** operators (that is, subexpressions E of expressions of the form **infer**. $D[e_1, \dots, e_m].c_j(E)$) for a given random trace. This semantics can be integrated to obtain the posterior distributions on random expressions on which queries depend. Then, we present a pseudo-deterministic *query semantics* which, given the distributions on random expressions, extracts the parameters of these distributions and computes the values of queries.

Restrictions of the semantics To handle conditioning and post-processing more easily, we impose three additional restrictions on Tabular tables, not enforced by the type system:

- Every expression in a conditioned **output** column must be a random draw from a primitive distribution—that is, for every **output** column ($c \triangleright x : T \ell$ **output** E) whose entry in the database is not **static**(?) or **inst**([?, ..., ?]), E must be of the form $D[e_1, \dots, e_m](E_1, \dots, E_n)$. The reason for this restriction is that the sampling semantics requires the density of the random expression E to be known. The density may not exist in general, and even when it does, deriving the density of an arbitrary random expression is a research problem in itself [Bhat et al., 2013] and is outside the scope of this dissertation.

This restriction is not significant in practice, because in nearly all applications of soft conditioning, the conditioning is performed on a value of a particular random draw rather than an arbitrary computation, and the default inference algorithm for Tabular is restricted to such models anyway.

- Likewise, to simplify identification of expressions inside the **infer** operator when calculating the map of marginals, we only allow the **infer** operator to be used at the top level—that is, in a column ($c \triangleright x : T \ell$ *viz* E), either $E = \mathbf{infer}.D[e_1, \dots, e_m].c_j(E')$ or E contains no subexpression of the form

infer. $D[e_1, \dots, e_m].c_j(E')$. This restriction could easily be removed by identifying expressions in **infer** by their positions in the top-level expressions, in addition to the names of corresponding columns—this would, however, be tedious, so we avoid doing that to keep the presentation clean.

An alternative would be to pre-transform the schema by moving all **infer** operators to separate columns.

- To simplify presentation, we only consider real-valued continuous distributions. However, the semantics could be easily extended to support discrete ones as well.

Note that the semantics presented in this section is independent of the inference engine used by the Tabular implementation and models supported by the semantics are not necessarily supported by the inference algorithm.

4.5.1 Evaluation Environments and Databases

In order to define the semantics, we need several evaluation environments, storing the values of already evaluated columns or local variables. As an environment is essentially an intermediate database, we use the same notation for environments as for databases—that is, δ is a schema-level environment consisting of maps τ of values of columns in individual tables. We also overload the symbol τ to denote an environment in which the identifiers are local variable names, rather than field names.

Non-nullable Values and Tabular Typing Environments:

$\tau ::= [x \mapsto \ell(V)^{i \in 1..n}]$	table-level environment with local identifiers
---	--

We write $\delta, (t \mapsto \tau)$ for the environment or database δ extended with table-level database τ assigned to t , and $\tau, (c \mapsto \ell(V))$ for the table-level environment or database τ with $\ell(V)$ assigned to c (and similarly for environments with local names).

4.5.2 Input Database Conformance

A full Core Tabular model, for which we can define the semantics, consists of a Core schema \mathbb{S} and a database DB . Obviously, the model is only well-defined if the database DB is a valid database for the schema \mathbb{S} —that is, it has values for all the **input** columns

and the conditioned **output** columns of \mathbb{S} , and all the values have the right types. Note that whether a given database conforms to a schema depends only on the type of the schema.

All **output** columns must have corresponding entries of the form **static**(V) or **inst**($[V_1, \dots, V_n]$) in the database; all the values in these entries can, however, be absent (recall that we denote an absent value by the wildcard “?”). Hence, an unconditioned **output** column has an entry in DB in which all the values are “?”.

We formalize the notion of well-formedness by the *conformance relation* $DB \models Sty$, stating that the database DB conforms to the schema type Sty . This relation is defined inductively; we start by presenting the rules deriving the auxiliary table-level judgment $(\tau, \rho_{sz}) \models_n Q$, which states that the table-level database τ , having n rows conforms, together with the table size map ρ_{sz} , to the table type Q .

Although values V are technically a distinct syntactic category from expressions E in the syntax of Tabular, in the typing judgments in the rest of this section, we will treat non-null values V as Tabular expressions and write $\Gamma \vdash^\ell V : T$ to mean that V is not “?” and has type T in Γ (at level ℓ) when considered as a Tabular expression.

In the rules below, we write $\rho_{sz}(T)$ for the type T with all table sizes of the form **sizeof**(t) substituted by corresponding entries in ρ_{sz} .

Conformance Rules for Table-Level Input Databases: $(\tau, \rho_{sz}) \models_n Q$

(CONF \square)

$(\tau, \rho_{sz}) \models_n \square$

(CONF STATIC INPUT)

$\tau(c) = \mathbf{static}(V) \quad \emptyset \vdash^{\mathbf{static}} V : \rho_{sz}(T)$

$(\tau, \rho_{sz}) \models_n Q \quad \neg \mathbf{det}(T)$

$(\tau, \rho_{sz}) \models_n (c \triangleright x : T \mathbf{static input}) :: Q$

(CONF INST INPUT)

$\tau(c) = \mathbf{inst}([V_0, \dots, V_{n-1}]) \quad \emptyset \vdash^{\mathbf{inst}} V_i : \rho_{sz}(T) \quad \forall i \in 0..n-1$

$(\tau, \rho_{sz}) \models_n Q$

$(\tau, \rho_{sz}) \models_n (c \triangleright x : T \mathbf{inst input}) :: Q$

(CONF STATIC OUTPUT)

$$\tau(c) = \mathbf{static}(V) \quad V = ? \vee \emptyset \vdash^{\mathbf{static}} V : \rho_{sz}(T)$$

$$(\tau, \rho_{sz}) \models_n Q \quad \neg \mathbf{det}(T)$$

$$\hline (\tau, \rho_{sz}) \models_n (c \triangleright x : T \mathbf{static output}) :: Q$$

(CONF INST OUTPUT)

$$\tau(c) = \mathbf{inst}([V_0, \dots, V_{n-1}]) \quad V_i = ? \vee \emptyset \vdash^{\mathbf{inst}} V_i : \rho_{sz}(T) \quad \forall i \in 0..n-1$$

$$(\tau, \rho_{sz}) \models_n Q$$

$$\hline (\tau, \rho_{sz}) \models_n (c \triangleright x : T \mathbf{inst output}) :: Q$$

The first rule (CONF \square) is obvious. The rule (CONF STATIC INPUT) requires that for every **static input** column in the table type, there must be a well-typed value (with the **static** tag) in the database τ . Static **input** columns cannot be deterministic, as already required by the type system (recall that types of top-level Tabular tables are supposed to be closed, and **static det** columns could be referenced in subsequent column types). Similarly, (CONF INST INPUT) states that each **inst-level input** column must have an **inst**-tagged entry in the database, being an array of n well-typed values. The (CONF STATIC OUTPUT) rule requires that the values on which **static output** columns are conditioned must be well-typed (that is, values corresponding to **static output** columns must either be wildcards “?” or values with matching types). Likewise, (CONF INST OUTPUT) requires all values on which particular rows of an **inst output** column are conditioned to be well-typed—in other words, for each row of such a column c , the corresponding value V_i in $\tau(c)$ can either be missing (denoted by ?) or be well-typed.

The top-level judgment $DB \models Sty$ is derived by the following rules:

Conformance Rules for Input Databases: $DB \models Sty$

(CONF SCHEMA \square)

$$\hline (\delta, \rho_{sz}) \models \square$$

(CONF SCHEMA TABLE)

$$\rho_{sz}(t) \in \mathbb{N} \quad (\delta(t), \rho_{sz}) \models_{\rho_{sz}(t)} Q \quad (\delta, \rho_{sz}) \models Sty$$

$$\hline (\delta, \rho_{sz}) \models (t : Q) :: Sty$$

The first rule (CONF SCHEMA \square) states that every database conforms to an empty

schema type. The (CONF SCHEMA TABLE) rule checks the conformance of the database stored in δ under t to the first table type Q in the given schema type and proceeds to check the conformance of the database to the rest of the schema type. Obviously, ρ_{sz} must store a valid size of the table stored at t .

4.5.3 Semantics of Probabilistic Models

We define the sampling semantics of Core Tabular, giving rise to the measure-theoretic semantics of random expressions. To save space, we assume the fixed map ρ_{sz} , storing sizes of tables in DB , and δ_{in} , being the map of tables in the input database, are globally accessible and do not mention it explicitly in the evaluation rules.

We say that a schema \mathbb{S} with database DB is *evaluated with trace s* if the values of all sampled random variables in the schema are fixed to the consecutive elements of the trace s (when sampling the required values for all rows of a given column before moving to the next column).

The key top-level judgment in the operational semantics is $DB \vdash \mathbb{S} \Downarrow_w^s \delta_{qry}$, which states that in schema \mathbb{S} with database DB , for every column c in table t with expression of the form **infer**. $D[e_1, \dots, e_m].c_j(E)$, $\delta_{qry}(t)(c) = \ell(V)$, where:

- If the column c is **static**, V is the value to which E evaluates if the entire schema \mathbb{S} is evaluated with the random trace s .
- If the column c is **inst**, $V = [V_0, \dots, V_{\rho_{sz}(t)-1}]$, such that for each $i \in 0.. \rho_{sz}(t) - 1$, V_i is the value to which E evaluates in the i -th row, if \mathbb{S} is evaluated with the random trace s .

We begin by showing the rules for evaluating Core expressions in unconditioned columns. The judgment $\delta; \tau; i \vdash E \Downarrow_w^s V$ says that in the schema-level environment δ and table-level environment τ , the expression E reduces in the i -th row of the table with trace s to a value V with weight w . The auxiliary judgment $\delta; \tau; i \vdash e \Downarrow V$ for indexed expressions is similar, except it has no trace or weight, as indexed expressions cannot contain random draws.

Expression evaluation: $\delta; \tau; i \vdash e \Downarrow V, \quad \delta; \tau; i \vdash E \Downarrow_w^s V$

(EVAL VAR STATIC)	(EVAL VAR INST)	(EVAL CONST)
$\tau(x) = \mathbf{static}(V)$	$\tau(x) = \mathbf{inst}([V_0, \dots, V_{n-1}])$	
$\delta; \tau; i \vdash x \Downarrow V$	$\delta; \tau; i \vdash x \Downarrow V_i$	$\delta; \tau; i \vdash s \Downarrow s$

(EVAL SIZEOF)	(EVAL INDEXED)	(EVAL Deref STATIC)
$\frac{\rho_{sz}(t) = n}{\delta; \tau; i \vdash \mathbf{sizeof}(t) \Downarrow n}$	$\frac{\delta; \tau; i \vdash e \Downarrow V}{\delta; \tau; i \vdash e \Downarrow_1^\square V}$	$\frac{\delta(t)(c) = \mathbf{static} V}{\delta; \tau; i \vdash t.c \Downarrow_1^\square V}$
(EVAL Deref INST)	(EVAL PRIM)	
$\frac{\delta; \tau; i \vdash E \Downarrow_w^s k}{\delta; \tau; i \vdash E : t.c \Downarrow_w^s V_k}$	$\frac{\delta; \tau; i \vdash E_j \Downarrow_{w_j}^{s_j} V_j \quad \forall j \in 1..n}{\delta; \tau; i \vdash g(E_1, \dots, E_n) \Downarrow_{w_1 \dots w_n}^{s_1 @ \dots @ s_n} \underline{g}(V_1, \dots, V_n)}$	
(EVAL IF TRUE)	(EVAL IF FALSE)	
$\frac{\delta; \tau; i \vdash E_1 \Downarrow_{w_1}^{s_1} \mathbf{true} \quad \delta; \tau; i \vdash E_2 \Downarrow_{w_2}^{s_2} V}{\delta; \tau; i \vdash \mathbf{if} E_1 \mathbf{then} E_2 \mathbf{else} E_3 \Downarrow_{w_1 w_2}^{s_1 @ s_2} V}$	$\frac{\delta; \tau; i \vdash E_1 \Downarrow_{w_1}^{s_1} \mathbf{false} \quad \delta; \tau; i \vdash E_3 \Downarrow_{w_2}^{s_2} V}{\delta; \tau; i \vdash \mathbf{if} E_1 \mathbf{then} E_2 \mathbf{else} E_3 \Downarrow_{w_1 w_2}^{s_1 @ s_2} V}$	
(EVAL ARRAY)	(EVAL INDEX) (where $j \in 0..n-1$)	
$\frac{\delta; \tau; i \vdash E_j \Downarrow_{w_j}^{s_j} V_j \quad \forall j \in 0..n-1}{\delta; \tau; i \vdash [E_0, \dots, E_{n-1}] \Downarrow_{w_0 \dots w_{n-1}}^{s_0 @ \dots @ s_{n-1}} [V_0, \dots, V_{n-1}]}$	$\frac{\delta; \tau; i \vdash E \Downarrow_{w_1}^{s_1} [V_0, \dots, V_{n-1}] \quad F \Downarrow_{w_2}^{s_2} j}{\delta; \tau; i \vdash E[F] \Downarrow_{w_1 w_2}^{s_1 @ s_2} V_j}$	
(EVAL ITER)		
$\frac{\delta; \tau; i \vdash e \Downarrow n \quad \delta; \tau, (x \mapsto \mathbf{static}(j)); i \vdash F \Downarrow_{w_j}^{s_j} V_j \quad \forall j \in 0..n-1}{\delta; \tau; i \vdash [\mathbf{for} x < e \rightarrow F] \Downarrow_{w w_0 \dots w_{n-1}}^{s @ s_0 @ \dots @ s_{n-1}} [V_0, \dots, V_{n-1}]}$		
(EVAL RANDOM)		
$\frac{\delta; \tau; i \vdash e_i \Downarrow V_i \quad \forall i \in 1..m \quad \delta; \tau; i \vdash E_j \Downarrow_{w_j}^{s_j} W_j \quad \forall j \in 1..n \quad \text{pdf}_{D[V_1, \dots, V_n]}(W_1, \dots, W_m, c) = w}{\delta; \tau; i \vdash D[e_1, \dots, e_m](E_1, \dots, E_n) \Downarrow_{w_1 \dots w_n w}^{s_1 @ \dots @ s_n @ [c]} c}$		

The rules (EVAL VAR STATIC) and (EVAL VAR INST) evaluate a variable x to its value in the (table-level) environment τ . If the variable is at **inst** level in the environment, the corresponding entry in τ must be an array, whose i -th element is returned. Similarly, (EVAL Deref STATIC) evaluates a reference $t.c$ to the value assigned to the identifier c in the table-level database assigned to t in the map δ ; (EVAL Deref INST) evaluates an **inst**-level reference $E : t.c$ to the k -th element of the array assigned to c in the table-level database assigned to t , where k is the value obtained by evaluating E . The expression E technically may contain a random draw, so its evaluation must be parametrised by a trace and a weight, which are then copied to the evaluation judgment for $E : t.c$. The rule (EVAL SIZEOF) simply applies the ρ_{sz} map to the table name t . In

the second assumption of (EVAL ITER), the value j of the local variable x is added to the environment at level **static**, regardless of whether the currently evaluated expression is **static** or **inst**, because this has the desired effect of simulating a substitution of j for x in F (we extend τ instead of using a substitution in order to preserve the conformance of the map τ to the typing environment, as defined later). All the other rules are standard for a first-order language, except that the traces and weights appearing in the assumptions of a rule must be combined in the judgment derived by this rule.

The (EVAL RANDOM) rule, which evaluates random draws for which there are no entries in the input database, evaluates the parameters of the given distribution and returns the first (and only) element c of the random trace, assumed to be the value drawn from the distribution D , together with the weight w being the density of D at c .

To reduce expressions in Tabular columns, we need to define two additional judgments: $\delta; \tau; i; V \vdash E \Downarrow_w^s W$ states that in the environment consisting of δ and τ , the expression E reduces in the i -th row to the value W under trace s , yielding weight w , if the corresponding entry in the database (at row i) is V , which may be $?$. If the expression is **static**, then i is expected to be 0, as the column has only one row. The judgment $\delta; \tau; \ell(V) \vdash E \Downarrow_w^s W$ is similar, except that it evaluates the entire column of a table; here, $\ell(V)$ is the entry in the database and if $\ell = \mathbf{inst}$, then both V and W are arrays, with one entry per row in the input and output databases.

Expression evaluation: $\delta; \tau; i; V \vdash E \Downarrow_w^s W, \quad \delta; \tau; \ell(V) \vdash E \Downarrow_w^s W$

(EVAL COND)

$$\begin{array}{l} \delta; \tau; i \vdash e_i \Downarrow V_i \quad \forall i \in 1..m \\ \delta; \tau; i \vdash E_j \Downarrow_{w_j}^{s_j} W_j \quad \forall j \in 1..n \\ \text{pdf}_{D[V_1, \dots, V_n]}(W_1, \dots, W_m, c) = w \\ \hline \delta; \tau; i; c \vdash D[e_1, \dots, e_m](E_1, \dots, E_n) \Downarrow_{w_1 \dots w_n}^{s_1 @ \dots @ s_n} c \end{array}$$

(EVAL SAMPLE)

$$\frac{\delta; \tau; i \vdash E \Downarrow_w^s V}{\delta; \tau; i; ? \vdash E \Downarrow_w^s V}$$

(EVAL STATIC)

$$\frac{\delta; \tau; 0; V \vdash E \Downarrow_w^s W}{\delta; \tau; \mathbf{static}(V) \vdash E \Downarrow_w^s W}$$

(EVAL INST)

$$\frac{\delta; \tau; i; V_i \vdash E \Downarrow_{w_i}^{s_i} W_i \quad \forall i \in 1..n}{\delta; \tau; \mathbf{inst}([V_1, \dots, V_n]) \vdash E \Downarrow_{w_1 \dots w_n}^{s_1 @ \dots @ s_n} [W_1, \dots, W_n]}$$

The rules (EVAL STATIC) and (EVAL INST) allow the derivation of the top-level judgment for expressions, $\delta; \tau; \ell(V) \vdash E \Downarrow_w^s W$, from the judgment $\delta; \tau; i; V \vdash E \Downarrow_w^s W$ evaluating individual rows. While (EVAL STATIC) just copies the result obtained by the latter judgment, (EVAL INST) combines the values obtained by evaluating E in different rows into an array.

The two rules (EVAL SAMPLE) and (EVAL COND) evaluate top-level expressions in single rows. The rule (EVAL SAMPLE) applies when the i -th row of the entry for the column with expression E in the database is “?”—that is, E is not conditioned. The expression E is then simply evaluated by the evaluation rules for unconditioned columns presented earlier.

The (EVAL COND) rule applies when the top-level expression is a random draw *conditioned* on the given value c . In this case, the random draw is evaluated to c , without consuming any part of the random trace other than those consumed by evaluating parameters, and the weight is set to the density of the distribution D (with the given parameters) at c , multiplied by the weights yielded by evaluating parameters (which must be equal to 1 unless at least one parameter is a nested random draw).

Next, we present the table evaluation rules. They derive the judgment $t; \delta; \tau' \vdash \mathbb{T} \Downarrow_w^s \tau; \tau_{qry}$, which states that in the environments δ and τ' , and given the (implicit) database (δ_{in}, ρ_{sz}) , the table \mathbb{T} named t evaluates with trace s to the pair of databases (τ, τ_{qry}) , such that

- Every expression in a random or deterministic column c of \mathbb{T} evaluates to the tagged value $\tau(c) = \ell(V)$ (where, again, V is an array of values of E in different rows if $\ell = \mathbf{static}$)
- Every random expression inside an **infer** operator in a column c of \mathbb{T} evaluates to $\tau_{qry}(c) = \ell(V)$

While the goal of the sampling semantics is to evaluate the expressions in **qry**-level columns, the values of random and deterministic columns also need to be computed (and added to the environment), because expressions in **infer** operators in subsequent tables may depend on them.

The rules for evaluating tables are as shown below. These rules use an auxiliary meta-language construct $\ell\langle t \rangle$, which denotes **static**(?) if $\ell = \mathbf{static}$ and **inst**([for $i <$

$\rho_{sz}(t) \rightarrow ?]$) if $\ell = \mathbf{inst}$ (where the **for**-loop is in the metalanguage). We also use the symbol $@$ for map concatenation.

Table Evaluation: $t; \delta; \tau' \vdash \mathbb{T} \Downarrow_w^s \tau; \tau_{qry}$

(EVAL EMPTY) (EVAL INPUT) (where $\neg \mathbf{qry}(T)$)	
$t; \delta; \tau' \vdash \square \Downarrow_1^\square \square$	$\frac{t; \delta; \tau', (x \mapsto \delta_{in}(t)(c)) \vdash \mathbb{T} \Downarrow_w^s \tau; \tau_{qry}}{t; \delta; \tau' \vdash (c \triangleright x : T \ell \mathbf{input} \varepsilon) :: \mathbb{T} \Downarrow_w^s [c \mapsto \delta_{in}(t)(c)] @ \tau; \tau_{qry}}$
(EVAL OUTPUT) (where $\neg \mathbf{qry}(T)$)	
$\delta; \tau'; \delta_{in}(t)(c) \vdash E \Downarrow_{w_1}^{s_1} V$	
$t; \delta; \tau', (x \mapsto \ell(V)) \vdash \mathbb{T} \Downarrow_{w_2}^{s_2} \tau; \tau_{qry}$	
$t; \delta; \tau' \vdash (c \triangleright x : T \ell \mathbf{output} E) :: \mathbb{T} \Downarrow_{w_1 w_2}^{s_1 @ s_2} [c \mapsto \ell(V)] @ \tau, \tau_{qry}$	
(EVAL LOCAL) (where $\neg \mathbf{qry}(T)$)	
$\delta; \tau'; \ell(t) \vdash E \Downarrow_{w_1}^{s_1} V$	
$t; \delta; \tau', (x \mapsto \ell(V)) \vdash \mathbb{T} \Downarrow_{w_2}^{s_2} \tau; \tau_{qry}$	
$t; \delta; \tau' \vdash (c \triangleright x : T \ell \mathbf{local} E) :: \mathbb{T} \Downarrow_{w_1 w_2}^{s_1 @ s_2} \tau; \tau_{qry}$	
(EVAL QUERY)	
$\mathbf{qry}(T)$	
$E = \mathbf{infer}.D[e_1, \dots, e_m].c_j(E')$	
$\delta; \tau'; \ell(t) \vdash E' \Downarrow_{w_1}^{s_1} V$	
$t; \delta; \tau' \vdash \mathbb{T} \Downarrow_{w_2}^{s_2} \tau; \tau_{qry}$	
$t; \delta; \tau' \vdash (c \triangleright x : T \ell \mathbf{viz} E) :: \mathbb{T} \Downarrow_{w_1 w_2}^{s_1 @ s_2} \tau; [c \mapsto \ell(V)] @ \tau_{qry}$	
(EVAL SKIP)	
$\mathbf{qry}(T)$	
$E \neq \mathbf{infer}.D[e_1, \dots, e_m].c_j(E')$	
$t; \delta; \tau' \vdash \mathbb{T} \Downarrow_w^s \tau; \tau_{qry}$	
$t; \delta; \tau' \vdash (c \triangleright x : T \ell \mathbf{viz} E) :: \mathbb{T} \Downarrow_w^s \tau; \tau_{qry}$	

The first rule, (EVAL EMPTY), is trivial. The rule (EVAL INPUT) simply copies a value from the input database to the corresponding entry in τ . The rule (EVAL OUTPUT) evaluates the deterministic or random expression E in an **output** column, stores its value in the environment τ' , evaluates the rest of the table and adds the labelled value obtained by evaluating E to the non-query map τ , obtained by evaluating the rest of the table. The weight of the given output databases is the product of weights yielded

by evaluating the expression (once for each row if it is at **inst** level) and evaluating the rest of the table. The following rule, (EVAL LOCAL), is similar, except that it does not append the value of the evaluated expression to τ' , as local columns are not exported. The rule (EVAL QUERY) applies if the expression E in the given column is of the form **infer**. $D[e_1, \dots, e_m].c_j(E')$; it evaluates the expression E' in the given environments and adds the resulting tagged value $\ell(V)$ to the query map τ_{qry} . Finally (EVAL SKIP) discards **qry**-level columns which are not of the form **infer**. $D[e_1, \dots, e_m].c_j(E')$ —these will be evaluated later by the query semantics.

Note that if an input database conforms to the schema type, all the values added to the environment τ' are not “?”.

At the end, we need to define two schema evaluating rules, deriving the top-level judgment $DB \vdash \mathbb{S} \Downarrow_w^s \delta_{qry}$:

Schema evaluation: $DB \vdash \mathbb{S} \Downarrow_w^s \delta_{qry}$

(EVAL SCHEMA EMPTY)	(EVAL SCHEMA TABLE)
$(\delta, \rho_{sz}) \vdash \Box \Downarrow_1^{\Box} \Box$	$ \begin{array}{c} t; \delta; \emptyset \vdash \mathbb{T} \Downarrow_{w_1}^{s_1} \tau_t; \tau_{tq} \\ \hline ((\delta, (t \rightarrow \tau_t)), \rho_{sz}) \vdash \mathbb{S} \Downarrow_{w_2}^{s_2} \delta_{qry} \\ \hline (\delta, \rho_{sz}) \vdash (t = \mathbb{T}) :: \mathbb{S} \Downarrow_{w_1 w_2}^{s_1 @ s_2} [t \rightarrow \tau_{tq}] @ \delta_{qry} \end{array} $

The rule (EVAL SCHEMA EMPTY) is obvious, while (EVAL SCHEMA TABLE) evaluates the given table \mathbb{T} with name t in the environment δ (the table-level environment, storing values of variables evaluated in the current table, is initially empty), which yields the non-query map τ_t and the query map τ_{tq} . The map τ_t is added to the environment δ before evaluating the rest of the schema and τ_{tq} is added to the output map. Note that we do not need to add τ_{tq} to the environment in which \mathbb{S} is evaluated, because the random expressions nested in queries cannot be directly referenced by other columns.

Example of Schema Evaluation To illustrate the random semantics with an example, let us consider a very simplified version of the Old Faithful model, reduced to Core form, which only models eruption durations and uses no indexing:

table faithful			
duration.Mean ▷ Mean	real!rnd	static output	GaussianFromMeanAndPrecision(0.0, 1.0)
duration.Prec ▷ Prec	real!rnd	static output	Gamma(1.0, 1.0)
duration ▷ duration	real!rnd	output	GaussianFromMeanAndPrecision(Mean, Prec)

Obviously, evaluating this model with any input database and valid random trace will yield the empty database, because the semantics returns a map of values of expressions in **infer** operators, and the above model contains no **infer**. Let us now add a **qry** level column, computing the mean of the posterior distribution of the mean duration.

table faithful			
duration.Mean \triangleright Mean	real!rnd	static output	GaussianFromMeanAndPrecision(0.0, 1.0)
duration.Prec \triangleright Prec	real!rnd	static output	Gamma(1.0, 1.0)
duration \triangleright duration	real!rnd	output	GaussianFromMeanAndPrecision(Mean, Prec)
posteriorMean \triangleright postMean	real!qry	static output	infer .Gaussian.mean(Mean)

Suppose that we have a tiny input database, which contains no entries for the **static** columns duration.Mean and duration.Prec and the duration column contains three entries, storing eruption durations in minutes:

ID	duration
0	4.0
1	5.0
2	4.0

More formally, the input database is $DB = (\delta_{in}, \rho_{sz})$, where $\delta_{in} = [\text{faithful} \mapsto [\text{duration.Mean} \mapsto \mathbf{static}(?), \text{duration.Prec} \mapsto \mathbf{static}(?), \text{duration} \mapsto \mathbf{inst}(4.0, 5.0, 4.0)]]$ and $\rho_{sz} = [\text{faithful} \mapsto 3]$.

It is easy to see that every valid trace in this model will be of length 2. Now, let us evaluate the model with trace $s = [3.0, 1.0]$. We want to compute δ_{qry} and w such that $DB \vdash \mathbb{S} \Downarrow_w^s \delta_{qry}$, where $\mathbb{S} = [(\text{faithful} \mapsto \mathbb{T})]$ and \mathbb{T} is the table shown above. We first need to compute τ , τ_{qry} and w such that $\text{faithful}; \emptyset; \emptyset \vdash \mathbb{T} \Downarrow_w^s \tau, \tau_{qry}$ (recall that the schema only has one table, so the entire trace s will be “consumed” when evaluating this table).

The first column of \mathbb{T} has visibility **output**, so it can be evaluated with the rule (EVAL OUTPUT):

(EVAL OUTPUT) (where $\neg \mathbf{qry}(T)$)

$$\emptyset; \emptyset; \delta_{in}(\text{faithful})(\text{duration.Mean}) \vdash E \Downarrow_{w_1}^{s_1} V$$

$$\text{faithful}; \emptyset; [(\text{Mean} \mapsto \mathbf{static}(V))] \vdash \mathbb{T}_2 \Downarrow_{w_2}^{s_2} \tau'; \tau_{qry}$$

$$\text{faithful}; \emptyset; \emptyset \vdash (\text{duration.Mean} \triangleright \text{Mean} : \mathbf{real!rnd static output } E) :: \mathbb{T}_2 \Downarrow_{w_1 w_2}^{s_1 @ s_2}$$

$$[\text{duration.Mean} \mapsto \mathbf{static}(V)] @ \tau', \tau_{qry}$$

where $s = s_1 @ s_2$ and $w = w_1 w_2$ and $E = \text{GaussianFromMeanAndPrecision}(0.0, 1.0)$ and \mathbb{T}_2 consists of all but the first column of the original table. In the first assumption,

we have $\delta_{in}(\text{faithful})(\text{duration.Mean}) = \mathbf{static}(?)$, as duration.Mean has no value in the input database. Hence, the first assumption can be derived with (EVAL STATIC), whose only assumption can in turn be derived with (EVAL SAMPLE), and the only assumption of (EVAL SAMPLE), $\emptyset; \emptyset; 0 \vdash E \Downarrow_{w_1}^{s_1} V$, by (EVAL RANDOM):

$$\begin{array}{l}
(\text{EVAL RANDOM}) \\
\emptyset; \emptyset; 0 \vdash 0.0 \Downarrow_1^\square 0.0 \\
\emptyset; \emptyset; 0 \vdash 1.0 \Downarrow_1^\square 1.0 \\
\frac{\text{pdf}_G(0.0, 1.0, 3.0) = w_1}{\emptyset; \emptyset; 0; ? \vdash G(0.0, 1.0) \Downarrow_{w_1}^{[3.0]} 3.0}
\end{array}$$

where $s_1 = [3.0]$ and $w_1 \approx 0.00443$ and G is an alias for `GaussianFromMeanAndPrecision`, used to save space.

Now we need to derive the second assumption and find w_2 , τ' and τ_{qry} . The second column is also an unconditioned **static output** column, so it can be evaluated with (EVAL OUTPUT):

$$\begin{array}{l}
(\text{EVAL OUTPUT}) \text{ (where } \neg \mathbf{qry}(T)) \\
\emptyset; [(\text{Mean} \mapsto \mathbf{static}(3.0))]; \mathbf{static}(?) \vdash E' \Downarrow_{w_2}^{s_2} V' \\
\text{faithful}; \emptyset; [(\text{Mean} \mapsto \mathbf{static}(3.0)), (\text{Prec} \mapsto \mathbf{static}(V'))] \vdash \mathbb{T}_3 \Downarrow_{w_2}^\square \tau''; \tau_{qry} \\
\hline
\text{faithful}; \emptyset; [(\text{Mean} \mapsto \mathbf{static}(3.0))] \vdash \\
(\text{duration.Prec} \triangleright \text{Prec} : \mathbf{real} ! \mathbf{rnd} \mathbf{static} \mathbf{output} E') :: \mathbb{T}_3 \Downarrow_{w_2 w_2'}^{s_2} \\
[\text{duration.Prec} \mapsto \mathbf{static}(V')] @ \tau'', \tau_{qry}
\end{array}$$

where $E' = \text{Gamma}(1.0, 1.0)$ and \mathbb{T}_3 contains the last two columns of the whole table \mathbb{T} . The first assumption of the above rule can, again, be derived by (EVAL RANDOM), via (EVAL STATIC) and (EVAL SAMPLE). We get $\emptyset; [(\text{Mean} \mapsto \mathbf{static}(3.0))]; 0; ? \vdash \text{Gamma}(1.0, 1.0) \Downarrow_{w_2'}^{[1.0]} 1.0$ and $w_2' \approx 0.36788$. Now we need to derive the second judgment in the above rule and find w_2'' and τ'' and τ_{qry} .

The third column is an **inst output** column, so it can again be evaluated by (EVAL OUTPUT) (where we write τ_{MP} for $[(\text{Mean} \mapsto \mathbf{static}(3.0)), (\text{Prec} \mapsto \mathbf{static}(1.0))]$ for conciseness):

$$\begin{array}{l}
(\text{EVAL OUTPUT}) \text{ (where } \neg \mathbf{qry}(T)) \\
\emptyset; \tau_{MP}; \mathbf{inst}([4.0, 5.0, 4.0]) \vdash E'' \Downarrow_{w_3}^\square V'' \\
\text{faithful}; \emptyset; \tau_{MP}, (\text{duration} \mapsto \mathbf{inst}(V'')) \vdash \mathbb{T}_4 \Downarrow_{w_3}^\square \tau'''; \tau_{qry} \\
\hline
\text{faithful}; \emptyset; \tau_{MP}, (\text{duration} \mapsto \mathbf{inst}(V'')) \vdash \\
(\text{duration} \triangleright \text{duration} : \mathbf{real} ! \mathbf{rnd} \mathbf{static} \mathbf{output} E'') :: \mathbb{T}_4 \Downarrow_{w_3 w_3'}^\square \\
[\text{duration} \mapsto \mathbf{inst}(V'')] @ \tau''', \tau_{qry}
\end{array}$$

where $E'' = \text{GaussianFromMeanAndPrecision}(\text{Mean}, \text{Prec})$ and $w_2'' = w_3' w_3''$ and \mathbb{T}_4

contains the last column of \mathbb{T} . To derive the first judgment and find V'' , we need to use the rule (EVAL INST):

(EVAL INST)

$$\frac{\begin{array}{l} \emptyset; \tau_{MP}; 0; 4.0 \vdash E'' \Downarrow_{\hat{w}_0}^{\square} W_0 \\ \emptyset; \tau_{MP}; 1; 5.0 \vdash E'' \Downarrow_{\hat{w}_1}^{\square} W_1 \\ \emptyset; \tau_{MP}; 2; 4.0 \vdash E'' \Downarrow_{\hat{w}_2}^{\square} W_2 \end{array}}{\emptyset; \tau_{MP}; \mathbf{inst}([4.0, 5.0, 4.0]) \vdash E'' \Downarrow_{\hat{w}_0 \hat{w}_1 \hat{w}_2}^{\square} [W_0, W_1, W_2]}$$

where $V'' = [W_0, W_1, W_2]$ and $w'_3 = \hat{w}_0 \hat{w}_1 \hat{w}_2$. Each of the three assumptions of (EVAL INST) can be derived with (EVAL COND), which in the case of the first judgment takes the following form:

(EVAL COND)

$$\frac{\begin{array}{l} \emptyset; \tau_{MP}; 0 \vdash \text{Mean} \Downarrow_1^{\square} 3.0 \\ \emptyset; \tau_{MP}; 0 \vdash \text{Prec} \Downarrow_1^{\square} 1.0 \\ pdf_G(3.0, 1.0, 4.0) = \hat{w}_0 \end{array}}{\emptyset; \tau_{MP}; 0; 4.0 \vdash G(\text{Mean}, \text{Prec}) \Downarrow_{\hat{w}_0}^{\square} 4.0}$$

where $\hat{w}_0 \approx 0.24197$. The first two assumptions can be derived easily with (QUERY VAR STATIC).

In the same way, we can derive the remaining two assumptions of (EVAL INST) and get $W_1 = 5.0$, $W_2 = 4.0$, $\hat{w}_1 \approx 0.05399$ and $\hat{w}_2 \approx 0.24197$. In the end, we get $\emptyset; \tau_{MP}; \mathbf{inst}([4.0, 5.0, 4.0]) \vdash E'' \Downarrow_{0.00316}^{\square} [4.0, 5.0, 4.0]$.

We still need to evaluate the last column containing the query, to derive the second judgment in the last use of (EVAL OUTPUT). This column can be evaluated by (EVAL QUERY):

(EVAL QUERY)

$$\frac{\begin{array}{l} E''' = \mathbf{infer.Gaussian.mean}(\hat{E}) \\ \emptyset; \tau_{MPD}; \mathbf{inst}([?, ?, ?]) \vdash \hat{E} \Downarrow_1^{\square} \hat{V} \\ \text{faithful}; \emptyset; \tau_{MPD} \vdash \square \Downarrow_1^{\square} \tau'''; \tau'_{qry} \end{array}}{\text{faithful}; \emptyset; \tau_{MPD} \vdash [(\text{posteriorMean} \triangleright \text{postMean} : \mathbf{real} ! \mathbf{qry} \text{ static output } E''')] \Downarrow_1^{\square} \tau'''; [\text{posteriorMean} \mapsto \mathbf{static}(\hat{V})] @ \tau'_{qry}}$$

where $\hat{E} = \text{Mean}$, $\tau_{MPD} = \tau_{MP}$, $(\text{duration} \mapsto \mathbf{inst}([4.0, 5.0, 4.0]))$. By (EVAL SAMPLE) and (EVAL VAR STATIC), we immediately get $\emptyset; \tau_{MPD}; \mathbf{inst}([?, ?, ?]) \vdash \text{Mean} \Downarrow_1^{\square} 3.0$. By (EVAL EMPTY), $\text{faithful}; \emptyset; \tau_{MPD} \vdash \square \Downarrow_w^s \square; \square$, so we have $\text{faithful}; \emptyset; \tau_{MPD} \vdash [(c \triangleright x : \mathbf{real} ! \mathbf{rnd} \text{ static output } E''')] \Downarrow_1^{\square} \square; [\text{posteriorMean} \mapsto \mathbf{static}(3.0)]$.

Therefore, we have derived the judgment $\text{faithful}; \emptyset; \emptyset \vdash \mathbb{T} \Downarrow_w^{[3.0, 1.0]} \tau, \tau_{qry}$, where $\tau = [(\text{Mean} \mapsto \mathbf{static}(3.0)), (\text{Prec} \mapsto \mathbf{static}(1.0)), (\text{duration} \mapsto \mathbf{inst}([4.0, 5.0, 4.0]))]$ and $\tau_{qry} = [\text{posteriorMean} \mapsto \mathbf{static}(3.0)]$ and $w = w_1 w'_2 w'_3 \approx 0.000005$.

By (EVAL SCHEMA TABLE), we get $DB \vdash \mathbb{S} \Downarrow_w^{[3.0, 1.0]} [\text{posteriorMean} \mapsto \mathbf{static}(3.0)]$, where w is as above. Note that while the output query map, containing just one entry for `posteriorMean`, does not depend on the observed data in the column `duration`, the weight w does.

Well-definedness of the Semantics If a schema \mathbb{S} is well-typed and the database DB conforms to its type Sty , then we expect that every map δ_{qry} to which the schema evaluates under some trace s is valid—that is, it contains an entry with a matching label for every occurrence of **infer** in a column of \mathbb{S} , all the values in this entry are well-typed and the array of values has size corresponding to the size of the table if the column in question is at **inst** level.

The lemma below shows that the sampling semantics is well-formed in that respect. We write $\text{dom}(\mathbb{S})$ for the set of names of tables in \mathbb{S} , $\mathbb{S}(t)$ for the table with name t in \mathbb{S} (if $t \in \text{dom}(\mathbb{S})$) and $\text{cols}(\mathbb{T})$ for the set of columns of \mathbb{T} .

Lemma 7 *If $\text{Core}(\mathbb{S})$ and $\emptyset \vdash \mathbb{S} : Sty$ and $(\delta_{in}, \rho_{sz}) \models Sty$ and $(\delta_{in}, \rho_{sz}) \vdash \mathbb{S} \Downarrow_w^s \delta_{qry}$, then for every $t \in \text{dom}(\mathbb{S})$ and $(c \triangleright x : T \ell \text{ viz } \mathbf{infer}.D[e_1, \dots, e_m].c_j(E')) \in \text{cols}(\mathbb{S}(t))$:*

- If $\ell = \mathbf{static}$, then $\delta_{qry}(t)(c) = \mathbf{static}(V)$ and $\emptyset \vdash^{\mathbf{static}} V : \mathbf{real} ! \mathbf{rnd}$.
- If $\ell = \mathbf{inst}$, then $\delta_{qry}(t)(c) = \mathbf{inst}([V_0, \dots, V_{\rho_{sz}(t)-1}])$ and $\emptyset \vdash^{\mathbf{inst}} V_i : \mathbf{real} ! \mathbf{rnd}$ for all $i \in 0.. \rho_{sz} - 1$.

Proof: In Appendix C. The proof makes use of an additional conformance relation $(\rho_{sz}; \delta; \tau) \models_n^{rd} \Gamma$, relating evaluation environments to typing environments, which is also defined in the appendix. ■

4.5.4 The Probability Measures on Random Expressions

We have already defined the sampling semantics of Tabular, associating an output database storing the values of expressions in **infer** queries, together with the corresponding weight, to each valid source of randomness, represented by a trace s . However, in order to define the query semantics, we need to compute the marginal distributions of such expressions, rather than individual weighted values. These marginals can be computed by integrating the weight of a trace (treated as a function), conditioned on the value of a given expression, with respect to the stock measure on program traces defined in Section 3.2.2.

In order to do so, we first define a function $\mathbf{P}_{(\mathbb{S}, DB)} : \mathbb{U} \rightarrow \mathbb{R}$ which maps a trace s to the weight yielded by evaluating the schema \mathbb{S} with database DB , or to 0 if s is not a valid trace in (\mathbb{S}, DB) :

$$\mathbf{P}_{(\mathbb{S}, DB)}(s) = \begin{cases} w & \text{if } DB \vdash \mathbb{S} \Downarrow_w^s \delta_{qry} \\ 0 & \text{otherwise} \end{cases}$$

We also need to define a function computing the value of a given random expression in an **infer** query for a given trace s . To make this a total function on traces, we introduce an exception **error**³, which is returned if a given trace does not lead to a value and which does not check against any type. The function $\mathbf{O}_{(\mathbb{S}, DB, t, c, i)} : \mathbb{U} \rightarrow \mathbb{R} \uplus \{\text{error}\}$ maps a trace s to the value of the random expression in a query in column c of table t in the i -th row (in case of **static** columns, the index i is discarded). If there is such a value, then it must be in \mathbb{R} , by the restriction that we only allow real-valued continuous random variables. If the trace is not valid, the exception **error** is returned.

$$\mathbf{O}_{(\mathbb{S}, DB, t, c, i)}(s) = \begin{cases} V & \text{if } DB \vdash \mathbb{S} \Downarrow_w^s \delta_{qry} \text{ and } \delta_{qry}(t)(c) = \mathbf{static}(V) \\ V_i & \text{if } DB \vdash \mathbb{S} \Downarrow_w^s \delta_{qry} \text{ and } \delta_{qry}(t)(c) = \mathbf{inst}([V_0, \dots, V_{n-1}]) \\ \text{error} & \text{otherwise} \end{cases}$$

These functions can only be used to define the marginal distributions if they are measurable. We do not give a detailed proof of measurability, but such a proof would effectively use the same ideas as the proofs of measurability of the functions \mathbf{P}_M and \mathbf{O}_M in Chapter 6. Recall that \mathcal{B} is the Borel σ -algebra on \mathbb{R} and that if (X, Σ_1) and (Y, Σ_2) are measurable spaces, a function $f : X \rightarrow Y$ is measurable Σ_1/Σ_2 if for all $B \in \Sigma_2$, $f^{-1}(B) \in \Sigma_1$.

Lemma 8 *If $\text{Core}(\mathbb{S})$ and $\emptyset \vdash \mathbb{S} : \text{Sty}$ and $DB \models \text{Sty}$ and $DB = (\delta_{in}, \rho_{sz})$, then*

- *The function $\mathbf{P}_{(\mathbb{S}, DB)}$ is measurable \mathcal{S}/\mathcal{B} .*
- *For every $t \in \text{dom}(\mathbb{S})$ and $(c \triangleright x : T \mathbf{static} \text{ viz } \mathbf{infer}.D[e_1, \dots, e_m].c_j(E')) \in \text{cols}(\mathbb{S}(t))$, the function $\mathbf{O}_{(\mathbb{S}, DB, t, c, 0)}$ is measurable $\mathcal{S}/\sigma(\mathcal{B} \cup \{\text{error}\})$.*
- *For every $t \in \text{dom}(\mathbb{S})$ and $(c \triangleright x : T \mathbf{inst} \text{ viz } \mathbf{infer}.D[e_1, \dots, e_m].c_j(E')) \in \text{cols}(\mathbb{S}(t))$ and $i \in 0..\rho_{sz}(t) - 1$, the function $\mathbf{O}_{(\mathbb{S}, DB, t, c, i)}$ is measurable $\mathcal{S}/\sigma(\mathcal{B} \cup \{\text{error}\})$.*

³To avoid confusion, we use the exception **error** to denote a computation not leading to a value in the probabilistic semantics and **fail** to represent inference failure in the query semantics

Proof: Similar to the proofs of Lemmas 43 and 44 in Chapter 6. ■

We can now define the distributions on random expressions in Tabular queries. Intuitively, an unnormalised marginal distribution of the expression E in a query in a **static** column c in table t of the schema \mathbb{S} will be the measure mapping a set $A \subseteq \mathbb{R}$ of possible values of E to the integral of the model density $\mathbf{P}_{(\mathbb{S}, DB)}$ over the space of traces, limited to traces for which the value of E is in A . Formally, the marginal is defined as follows;

$$\text{marg}((\mathbb{S}, DB), t, c)(A) = \int \mathbf{P}_{(\mathbb{S}, DB)}(s) [(\mathbf{O}_{(\mathbb{S}, DB, t, c, 0)}(s)) \in A] ds$$

The marginal distribution of an expression E in the i -th row of the **inst**-level column c is defined similarly:

$$\text{marg}((\mathbb{S}, DB), t, c, i)(A) = \int \mathbf{P}_{(\mathbb{S}, DB)}(s) [(\mathbf{O}_{(\mathbb{S}, DB, t, c, i)}(s)) \in A] ds$$

Strictly speaking, the measure $\text{marg}((\mathbb{S}, DB), t, c)(A)$ is a restriction of the measure mapping $A \in \sigma(\mathcal{B} \cup \{\text{error}\})$ to $\int \mathbf{P}_{(\mathbb{S}, DB)}(s) [(\mathbf{O}_{(\mathbb{S}, DB, t, c, 0)}(s)) \in A] ds$ to the set \mathbb{R} of real numbers (without error), and similarly for $\text{marg}((\mathbb{S}, DB), t, c, i)(A)$.

The above marginals are unnormalised. Obviously, they can only be normalised, to yield valid probability distributions, if they are non-zero and finite. To guarantee that the marginals are positive, it is enough to assume that the integral of the density $\mathbf{P}_{(\mathbb{S}, DB)}$ is positive (or, in other words, that the program consisting of \mathbb{S} and DB has positive success probability), which is a standard assumption in the analysis of probabilistic models.

Assumption 1 *For every \mathbb{S} , DB such that $\text{Core}(\mathbb{S}), \emptyset \vdash \mathbb{S} : \text{Sty}$ and $DB \models \text{Sty}$, we have $\int \mathbf{P}_{(\mathbb{S}, DB)}(s) ds > 0$*

This assumption is sufficient to guarantee that all the marginals are non-zero:

Lemma 9 *Let \mathbb{S} , δ_{in} and ρ_{sz} be such that $\text{Core}(\mathbb{S}), \emptyset \vdash \mathbb{S} : \text{Sty}$ and $(\delta_{in}, \rho_{sz}) \models \text{Sty}$. Then, given Assumption 1, for every $t \in \text{dom}(\mathbb{S})$ and $(c \triangleright x : T \ell \text{ viz } \text{infer}.D[e_1, \dots, e_m].c_j(E')) \in \text{cols}(\mathbb{S}(t))$:*

- If $\ell = \text{static}$, then $\text{marg}((\mathbb{S}, (\delta_{in}, \rho_{sz})), t, c)(\mathbb{R}) > 0$.
- If $\ell = \text{inst}$, then $\text{marg}((\mathbb{S}, (\delta_{in}, \rho_{sz})), t, c, i)(\mathbb{R}) > 0$ for every $i \in 0..\rho_{sz}(t) - 1$.

Proof:

If $\ell = \mathbf{static}$, then by Lemma 7, for every s such that $(\delta_{in}, \rho_{sz}) \vdash \mathbb{S} \Downarrow_w^s \delta_{qry}$, we have $\delta_{qry}(t)(c) = \mathbf{static}(V)$ and $\emptyset \vdash^{\mathbf{static}} V : \mathbf{real} ! \mathbf{rnd}$, the latter judgment implying $V \in \mathbb{R}$. Hence, $\mathbf{O}_{(\mathbb{S}, DB, t, c, 0)}(s) \in \mathbb{R}$. This implies that $\mathbf{O}_{(\mathbb{S}, DB, t, c, 0)}(s) \in \mathbb{R}$ whenever $\mathbf{P}_{(\mathbb{S}, DB)}(s) > 0$, so $\text{marg}((\mathbb{S}, DB), t, c)(\mathbb{R}) = \int \mathbf{P}_{(\mathbb{S}, DB)}(s) ds > 0$ by Assumption 1.

The reasoning in the case $\ell = \mathbf{inst}$ is similar. ■

We also assume that the integral of the density $\mathbf{P}_{(\mathbb{S}, DB)}(s)$ is finite:

Assumption 2 *For every \mathbb{S} , DB such that $\text{Core}(\mathbb{S})$, $\emptyset \vdash \mathbb{S} : \text{Sty}$ and $DB \models \text{Sty}$, we have $\int \mathbf{P}_{(\mathbb{S}, DB)}(s) ds < \infty$.*

This assumption is necessary to avoid some degenerate cases of programs having an infinite normalisation constant and thus not defining probability measures on columns. For instance, consider the following Tabular program \mathbb{S} :

table t			
c ▷ x	real!rnd	static output	Gamma(0.5, 1)
d ▷ y	real!rnd	static output	Gaussian(0.0, (sqrt(x)exp(x)) [^] (-2)))

where sqrt and exp are primitive functions returning, respectively, the square root and the natural exponential of the given argument, and a corresponding database $DB = [t \mapsto [(c \mapsto \mathbf{static}(?)), (d \mapsto \mathbf{static}(0.0))]]$.

This program first draws a value from $\text{Gamma}(0.5, 1)$ and then observes that the draw from $\text{Gaussian}(0.0, (\sqrt{x} e^x)^{-2})$, where x is the outcome of the first random draw, returned 0. Recall that the density of the Gaussian distribution with mean μ and variance σ^2 is:

$$\text{pdf}_{\text{Gaussian}}(\mu, \sigma^2, y) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y-\mu)^2}{2\sigma^2}}$$

and the density of the Gamma distribution parametrised by the shape α and scale θ is:

$$\text{pdf}_{\text{Gamma}}(\alpha, \theta, y) = \frac{y^{\alpha-1} e^{-\frac{y}{\theta}}}{\Gamma(\alpha)\theta^\alpha}$$

for $y > 0$, where Γ is the gamma function, which is positive for positive arguments and satisfies $\Gamma(n) = (n-1)!$ for all positive integers n .

Obviously, each trace leading to a positive weight in the above program consists of just one real number c . In this case, the density of the program is

$$\mathbf{P}_{(\mathbb{S}, DB)}([c]) = \frac{c^{-0.5} e^{-c}}{\Gamma(0.5)} \frac{1}{\sqrt{2\pi(c^{0.5} e^c)^{-2}}} = \frac{1}{\Gamma(0.5)\sqrt{2\pi}}$$

if $c > 0$ and $\mathbf{P}_{(\mathbb{S}, DB)}([c]) = 0$ otherwise (because then c is outside the support of Gamma). Thus, the integral of the density $\mathbf{P}_{(\mathbb{S}, DB)}$ over the space of all traces is:

$$\int \mathbf{P}_{(\mathbb{S}, DB)}(s) ds = \int_{\mathbb{R}_+} \frac{1}{\Gamma(0.5)\sqrt{2\pi}} dc = \infty$$

Hence, we have $\text{marg}((\mathbb{S}, DB), t, c)(\mathbb{R}) = \int_{\mathbb{R}_+} \frac{1}{\Gamma(0.5)\sqrt{2\pi}} [\text{pdf}_{\text{Gamma}}(0.5, 1, c) \in \mathbb{R}] dc = \int_{\mathbb{R}_+} \frac{1}{\Gamma(0.5)\sqrt{2\pi}} dc = \infty$ and $\text{marg}((\mathbb{S}, DB), t, d)(\mathbb{R}) = \int_{\mathbb{R}_+} \frac{1}{\Gamma(0.5)\sqrt{2\pi}} [0.0 \in \mathbb{R}] dc = \int_{\mathbb{R}_+} \frac{1}{\Gamma(0.5)\sqrt{2\pi}} dc = \infty$, and so the program does not define marginal probability distributions on the table columns.

The two above assumptions ensure that the unnormalised marginals are finite measures:

Lemma 10 *If $\text{Core}(\mathbb{S})$ and $\emptyset \vdash \mathbb{S} : \text{Sty}$ and $(\delta_{in}, \rho_{sz}) \models \text{Sty}$, then for every $t \in \text{dom}(\mathbb{S})$ and $(c \triangleright x : T \ell \text{ viz } \mathbf{infer}.D[e_1, \dots, e_m].c_j(E')) \in \text{cols}(\mathbb{S}(t))$:*

- If $\ell = \mathbf{static}$, then $\text{marg}((\mathbb{S}, DB), t, c)(\mathbb{R}) < \infty$.
- If $\ell = \mathbf{inst}$, then $\text{marg}((\mathbb{S}, DB), t, c, i)(\mathbb{R}) < \infty$ for every $i \in 0..\rho_{sz}(t) - 1$.

Proof: Follows immediately from Assumption 2. ■

We can now normalise the marginal measures to obtain probability distributions on random expressions in **infer** operators. If column c in table t is **static**, the distribution of the expression in the query in c is defined as:

$$\widehat{\text{marg}}((\mathbb{S}, DB), t, c)(A) \triangleq \frac{\text{marg}((\mathbb{S}, DB), t, c)(A)}{\text{marg}((\mathbb{S}, DB), t, c)(\mathbb{R})}$$

If c is an **inst**-level column, and $0 \leq i < \rho_{sz}(t)$, the distribution of the expression in the i -th row of column c is:

$$\widehat{\text{marg}}((\mathbb{S}, DB), t, c, i)(A) \triangleq \frac{\text{marg}((\mathbb{S}, DB), t, c, i)(A)}{\text{marg}((\mathbb{S}, DB), t, c, i)(\mathbb{R})}$$

Lemma 11 *If $\text{Core}(\mathbb{S})$ and $\emptyset \vdash \mathbb{S} : \text{Sty}$ and $(\delta_{in}, \rho_{sz}) \models \text{Sty}$, then for every $t \in \text{dom}(\mathbb{S})$ and $(c \triangleright x : T \ell \text{ viz } \mathbf{infer}.D[e_1, \dots, e_m].c_j(E')) \in \text{cols}(\mathbb{S}(t))$:*

- If $\ell = \mathbf{static}$, then $\widehat{\text{marg}}((\mathbb{S}, DB), t, c)$ is a probability measure on \mathcal{B} .
- If $\ell = \mathbf{inst}$, then $\widehat{\text{marg}}((\mathbb{S}, DB), t, c, i)$ is a probability measure on \mathcal{B} for every $i \in 0..\rho_{sz}(t) - 1$.

Proof: Follows from Lemma 9 and Lemma 10. ■

4.5.5 Semantics of Queries

With the definitions of marginals in place, we can define the semantics of Tabular queries. This semantics will compute the values of all **qry** columns in a schema given the marginal distributions of random expressions referenced in queries, as defined in the previous section.

We begin by defining a function $\sigma(\mathbb{S}, DB)$, mapping a schema and a database to a map of marginal distributions of expressions in **infer** queries. This map σ will assign to each table name in \mathbb{S} a table-level map η , mapping each global column name to either a measure or an array of measures, depending on whether the given column is **static** or **inst**.

Databases, Tables, Attributes, and Values:

$\sigma ::= [t_i \mapsto \eta_i^{i \in 1..n}]$	schema-level marginal map
$\eta ::= [c_i \mapsto g_i^{i \in 1..m}]$	table-level marginal map
$g ::= v \mid [v_0, \dots, v_{n-1}]$	marginal map entry

We first define two auxiliary functions returning sets of names of **static** and **inst** query columns with **infer** constructs:

$$\text{qry_cols_static}(\mathbb{T}) \triangleq \{c \mid (c \triangleright x : T \text{ static viz infer.D}[e_1, \dots, e_m].c_j(E')) \in \text{cols}(\mathbb{T})\}$$

$$\text{qry_cols_inst}(\mathbb{T}) \triangleq \{c \mid (c \triangleright x : T \text{ inst viz infer.D}[e_1, \dots, e_m].c_j(E')) \in \text{cols}(\mathbb{T})\}$$

We also define two functions `marg_static` and `marg_inst` which return the marginal (or map of marginals) corresponding to the given column.

$$\text{marg_static}(\mathbb{S}, DB, t, c) \triangleq \widehat{\text{marg}}((\mathbb{S}, DB), t, c)$$

$$\text{marg_inst}(\mathbb{S}, (\delta_{in}, \rho_{sz}), t, c) \triangleq [\text{for } i < \rho_{sz}(t) \rightarrow \widehat{\text{marg}}((\mathbb{S}, DB), t, c, i)]$$

Note that the **for**-loop in the second definition is in the meta-language. We can now define the map $\sigma(\mathbb{S}, DB)$ of marginals in the model consisting of \mathbb{S} and DB :

$$\sigma(\mathbb{S}, DB) \triangleq [(t \mapsto [c \mapsto \text{marg_static}(\mathbb{S}, DB, t, c)^{c \in \text{qry_cols_static}(\mathbb{S}(t))}] @ [c \mapsto \text{marg_inst}(\mathbb{S}, DB, t, c)^{c \in \text{qry_cols_inst}(\mathbb{S}(t))}])^{t \in \text{dom}(\mathbb{S})}]$$

Example of marginal map computation Let us now revisit the simplified Old Faithful example to show how the map of marginals is constructed. To simplify computations, we restrict the model even further, by assuming the precision is fixed. This results in the schema \mathbb{S} consisting of a single table \mathbb{T} of the following form:

table faithful			
duration.Mean \triangleright Mean	real!rnd	static output	GaussianFromMeanAndPrecision(0.0, 1.0)
duration \triangleright duration	real!rnd	output	GaussianFromMeanAndPrecision(Mean, 1.0)
posteriorMean \triangleright postMean	real!qry	static output	infer.Gaussian.mean(Mean)

Let the input database be $DB = (\delta_{in}, \rho_{sz})$, where $\delta_{in} = [\text{faithful} \mapsto [\text{duration.Mean} \mapsto \text{static}(\text{?}), \text{duration} \mapsto \text{inst}(4.0, 5.0, 4.0)]]$ and $\rho_{sz} = [\text{faithful} \mapsto 3]$.

The only **infer** operator is in the last column, so we have $\sigma(\mathbb{S}, DB) = [(\text{faithful} \mapsto [\text{posteriorMean} \mapsto \text{static}(\widehat{\text{marg}}((\mathbb{S}, DB), \text{faithful}, \text{posteriorMean}))])]$. We can easily shown that the unnormalised marginal $\text{marg}((\mathbb{S}, DB), \text{faithful}, \text{posteriorMean})$ is:

$$\begin{aligned}
& \text{marg}((\mathbb{S}, DB), \text{faithful}, \text{posteriorMean})(A) \\
&= \int \mathbf{P}_{(\mathbb{S}, DB)}(s) [(\mathbf{O}_{(\mathbb{S}, DB), \text{faithful}, \text{posteriorMean}, 0)}(s)) \in A] ds \\
&= \int \text{pdf}_G(0, 1, s_1) \text{pdf}_G(s_1, 1, 4) \text{pdf}_G(s_1, 1, 5) \text{pdf}_G(s_1, 1, 4) [s_1 \in A] \lambda(ds_1) \\
&= \int_A \text{pdf}_G(0, 1, s_1) \text{pdf}_G(s_1, 1, 4) \text{pdf}_G(s_1, 1, 5) \text{pdf}_G(s_1, 1, 4) \lambda(ds_1)
\end{aligned}$$

where we write G for GaussianFromMeanAndPrecision, to save space. We can see that the normalised version of the above function is the posterior distribution of durationMean, which can in this case (a conjugate Gaussian model with fixed precision) be analytically computed. We have:

$$\widehat{\text{marg}}((\mathbb{S}, DB), \text{faithful}, \text{posteriorMean}) = G(3.25, 4)$$

where we write G for the Gaussian distribution with mean and precision parameters. Hence, the map of marginals is:

$$\sigma(\mathbb{S}, DB) = [(\text{faithful} \mapsto [\text{posteriorMean} \mapsto \text{static}(G(3.25, 4))])]$$

Query evaluation We can now present the rules computing the output database, consisting of values of queries.

We begin by presenting rules for deterministically evaluating Core model expressions without random draws. The judgment $\delta; \tau; i \vdash E \Downarrow V$ means that the expression E evaluates deterministically to the value V in the environment consisting of a schema-level map δ and table-level map τ , if we are currently evaluating the i -th row. The

rules below are essentially the rules for random expression evaluation with scores, traces and the (EVAL RANDOM) case removed. In fact, the judgment $\delta; \tau; i \vdash E \Downarrow V$ could be defined as holding if and only if $\delta; \tau; i \vdash E \Downarrow_1 V$ hold, but for clarity we define it separately by the following inductive rules:

Deterministic expression evaluation: $\delta; \tau; i \vdash E \Downarrow V$

<hr/>		
(QUERY VAR STATIC)	(QUERY VAR INST)	(QUERY CONST)
$\frac{\tau(x) = \mathbf{static}(V)}{\delta; \tau; i \vdash x \Downarrow V}$	$\frac{\tau(x) = \mathbf{inst}([V_0, \dots, V_{n-1}])}{\delta; \tau; i \vdash x \Downarrow V_i}$	$\frac{}{\delta; \tau; i \vdash s \Downarrow s}$
(QUERY PRIM)	(QUERY IF TRUE)	
$\frac{\delta; \tau; i \vdash E_j \Downarrow V_j \quad \forall j \in 1..n}{\delta; \tau; i \vdash g(E_1, \dots, E_n) \Downarrow \underline{g}(V_1, \dots, V_n)}$	$\frac{\delta; \tau; i \vdash E_1 \Downarrow \mathbf{true} \quad \delta; \tau; i \vdash E_2 \Downarrow V}{\delta; \tau; i \vdash \mathbf{if } E_1 \mathbf{ then } E_2 \mathbf{ else } E_3 \Downarrow V}$	
(QUERY IF FALSE)	(QUERY ARRAY)	
$\frac{\delta; \tau; i \vdash E_1 \Downarrow \mathbf{false} \quad \delta; \tau; i \vdash E_3 \Downarrow V}{\delta; \tau; i \vdash \mathbf{if } E_1 \mathbf{ then } E_2 \mathbf{ else } E_3 \Downarrow V}$	$\frac{\delta; \tau; i \vdash E_j \Downarrow V_j \quad \forall j \in 0..n-1}{\delta; \tau; i \vdash [E_0, \dots, E_{n-1}] \Downarrow [V_0, \dots, V_{n-1}]}$	
(QUERY INDEX) (where $j \in 0..n-1$)	(QUERY ITER)	
$\frac{\delta; \tau; i \vdash E \Downarrow [V_0, \dots, V_{n-1}] \quad \delta; \tau; i \vdash F \Downarrow j}{\delta; \tau; i \vdash E[F] \Downarrow V_j}$	$\frac{\delta; \tau; i \vdash e \Downarrow n \quad \delta; \tau, (x \mapsto \mathbf{static}(j)); i \vdash F \Downarrow V_j \quad \forall j \in 0..n-1}{\delta; \tau; i \vdash [\mathbf{for } x < e \rightarrow F] \Downarrow [V_0, \dots, V_{n-1}]}$	
(QUERY Deref STATIC)	(QUERY Deref INST)	(QUERY SIZEOF)
$\frac{\delta(t)(c) = \mathbf{static}(V)}{\delta; \tau; i \vdash t.c \Downarrow V}$	$\frac{\delta; \tau; i \vdash E \Downarrow k \quad \delta(t)(c) = \mathbf{inst}([V_0, \dots, V_{\rho_{sz}(t)-1}])}{\delta; \tau; i \vdash E : t.c \Downarrow V_k}$	$\frac{\rho_{sz}(t) = n}{\delta; \tau; i \vdash \mathbf{sizeof}(t) \Downarrow n}$
<hr/>		

The type system does not guarantee that inference will succeed. If, for example, the user defines a query $\mathbf{infer}.D[e_1, \dots, e_m].c_j(E')$ in which the distribution D does not match the marginal distribution of E' , then the query is not well-defined and the “optimal” value of c_j may be infinite. Because of this possibility of failure, a variable can evaluate to an exception `fail` in the query semantics. Hence, we need to extend above the set of deterministic evaluation rules with the standard exception-handling rules shown below. In the remainder of this section, we will denote by G the *generalised* values, which can be values V or `fail`. Note that the type system guarantees that only expressions in **qry** columns can evaluate to `fail`.

Evaluation of Erroneous Expressions: $\delta; \tau; i \vdash E \Downarrow G$

<hr/>	
(QUERY INDEX FAIL1)	(QUERY INDEX FAIL2)
$\delta; \tau; i \vdash E \Downarrow [V_0, \dots, V_{n-1}]$	$\delta; \tau; i \vdash E \Downarrow \text{fail}$
$\delta; \tau; i \vdash F \Downarrow \text{fail}$	$\delta; \tau; i \vdash E[F] \Downarrow \text{fail}$
$\delta; \tau; i \vdash E[F] \Downarrow \text{fail}$	
(QUERY IF FAIL)	(QUERY Deref INST FAIL)
$\delta; \tau; i \vdash E_1 \Downarrow \text{fail}$	$\delta; \tau; i \vdash E \Downarrow \text{fail}$
$\delta; \tau; i \vdash \text{if } E_1 \text{ then } E_2 \text{ else } E_3 \Downarrow \text{fail}$	$\delta; \tau; i \vdash E : t.c \Downarrow \text{fail}$
<hr/>	

We assume that **fail** (unlike **error** in the random semantics) checks against any type in space **qry**:

Typing Rules for Generalised Values: $\Gamma \vdash^{pc} G : T$

(FAIL)
$\Gamma \vdash T \quad \mathbf{qry}(T)$
$\Gamma \vdash^{pc} \text{fail} : T$

We can now define the table-level query evaluation rules. They derive the judgment $t; \delta_{in}; \delta; \tau'; \eta \vdash \mathbb{T} \Downarrow \tau$, which states that the table \mathbb{T} with name t , together with the (implicit) input database (δ_{in}, ρ_{sz}) , in the environment consisting of the table-level map τ' , schema-level map δ and a table-level map of marginals η , reduces to a table-level map τ , containing the values of **qry**-level columns.

In the rest of this section, we overload the notation $D[V_1, \dots, V_m](y_1, \dots, y_n)$ to denote the probability measure corresponding to the distribution D with given arguments and hyperparameters. Recall that we write $\|v_1 - v_2\|$ for the variational norm of two measures v_1 and v_2 on (Ω, Σ) , defined as $\|v_1 - v_2\| = \sup_{A \in \Sigma} |v_1(A) - v_2(A)|$. We assume that $\arg \min_{y_1, \dots, y_n} f(y_1, \dots, y_n)$ returns an n -tuple of exceptions **fail** if the minimum does not exist and that if there are multiple combinations of arguments y_1, \dots, y_n minimising f , $\arg \min$ returns one of them nondeterministically.

Table Evaluation: $t; \delta_{in}; \delta; \tau'; \eta \vdash \mathbb{T} \Downarrow \tau$

(VAL EMPTY)	(VAL INPUT) (where $\text{space}(T) \neq \mathbf{rnd}$)
	$t; \delta_{in}; \delta; \tau', (x \rightarrow \delta_{in}(t)(c)); \eta \vdash \mathbb{T} \Downarrow \tau$
$t; \delta_{in}; \delta; \tau'; \eta \vdash [] \Downarrow []$	$t; \delta_{in}; \delta; \tau'; \eta \vdash (c \triangleright x : T \ell \text{ input } \varepsilon) :: \mathbb{T} \Downarrow \tau @ [c \mapsto \delta_{in}(t)(c)]$

(VAL RANDOM) (where $\text{space}(T) = \mathbf{rnd}$)

$$\frac{t; \delta_{in}; \delta; \tau'; \eta \vdash \mathbb{T} \Downarrow \tau}{t; \delta_{in}; \delta; \tau'; \eta \vdash (c \triangleright x : T \text{ viz } M) :: \mathbb{T} \Downarrow \tau}$$

(VAL QUERY STATIC)

$$\frac{\begin{array}{l} t; \delta; \tau'; 0 \vdash e_k \Downarrow s_k \quad \forall k \in 1..m \\ (G_1, \dots, G_n) = \arg \min_{y_1, \dots, y_n} \|D[s_1, \dots, s_m](y_1, \dots, y_n) - \eta(c)\| \\ t; \delta_{in}; \delta; \tau', (x \rightarrow \mathbf{static}(G_j)); \eta \vdash \mathbb{T} \Downarrow \tau \end{array}}{t; \delta_{in}; \delta; \tau'; \eta \vdash (c \triangleright x : T \mathbf{static} \text{ viz } (\mathbf{infer}.D[e_1, \dots, e_m].c_j(E'))) :: \mathbb{T} \Downarrow \tau@[c \mapsto \mathbf{static}(G_j)]}$$

(VAL QUERY INST)

$$\frac{\begin{array}{l} t; \delta; \tau'; i \vdash e_k \Downarrow s_{i,k} \quad \forall i \in 0..(\rho_{sz}(t) - 1), k \in 1..m \\ (G_{i,1}, \dots, G_{i,n}) = \arg \min_{y_1, \dots, y_n} \|D[s_{i,1}, \dots, s_{i,m}](y_1, \dots, y_n) - \eta(c)[i]\| \quad \forall i \in 0..(\rho_{sz}(t) - 1) \\ t; \delta_{in}; \delta; \tau', (x \rightarrow \mathbf{inst}([G_{0,j}, \dots, G_{\rho_{sz}(t)-1,j}])); \eta \vdash \mathbb{T} \Downarrow \tau \end{array}}{t; \delta_{in}; \delta; \tau'; \eta \vdash (c \triangleright x : T \mathbf{inst} \text{ viz } (\mathbf{infer}.D[e_1, \dots, e_m].c_j(E'))) :: \mathbb{T} \Downarrow \tau@[c \mapsto \mathbf{inst}([G_{0,j}, \dots, G_{\rho_{sz}(t)-1,j}])}]}$$

(VAL QUERYORDET STATIC) (where $\text{space}(T) \neq \mathbf{rnd}, E \neq \mathbf{infer}.D[e_1, \dots, e_m].c_j(E')$)

$$\frac{\begin{array}{l} t; \delta; \tau'; 0 \vdash E \Downarrow G \\ t; \delta_{in}; \delta; \tau', (x \rightarrow \mathbf{static}(G)); \eta \vdash \mathbb{T} \Downarrow \tau \end{array}}{t; \delta_{in}; \delta; \tau'; \eta \vdash (c \triangleright x : T \mathbf{static} \text{ viz } E) :: \mathbb{T} \Downarrow \tau@[c \mapsto \mathbf{static}(G)]}$$

(VAL QUERYORDET INST) (where $\text{space}(T) \neq \mathbf{rnd}, E \neq \mathbf{infer}.D[e_1, \dots, e_m].c_j(E')$)

$$\frac{\begin{array}{l} t; \delta; \tau'; i \vdash E \Downarrow G_i \quad \forall i \in 0..(\rho_{sz}(t) - 1) \\ t; \delta_{in}; \delta; \tau', (x \rightarrow \mathbf{inst}([G_0, \dots, G_{\rho_{sz}(t)-1}])); \eta \vdash \mathbb{T} \Downarrow \tau \end{array}}{t; \delta_{in}; \delta; \tau'; \eta \vdash (c \triangleright x : T \mathbf{inst} \text{ viz } E) :: \mathbb{T} \Downarrow \tau@[c \mapsto \mathbf{inst}([G_0, \dots, G_{\rho_{sz}(t)-1}])}]}$$

The base rule (VAL EMPTY) is, as usual, trivial. The (VAL INPUT) rule just copies an input value from the input database to the output map; it also assigns this value to x in the recursive call. Note that (VAL INPUT) only applies if the input column was not marked as **rnd**—otherwise, the given value is not assumed to be part of the output database and subsequent non-**rnd** values cannot depend on it. The rule (VAL RANDOM) simply discards random columns in the given table.

The (VAL QUERY STATIC) rule evaluates a query of the form $\mathbf{infer}.D[e_1, \dots, e_m].c_j(E')$ in a **static** column. The rule evaluates the hyperparameters e_1, \dots, e_m to s_1, \dots, s_m (which are expected to be scalar values) and looks up the marginal distribution of the expression E' in the marginal map η . It then tries to fit a distribution of the form $D[s_1, \dots, s_m](y_1, \dots, y_n)$ to the distribution $\eta(c)$ of E' , by minimising the variational

norm of $D[s_1, \dots, s_m](y_1, \dots, y_n)$ and $\eta(c)$ with respect to y_1, \dots, y_n . Typically, the posterior distribution $\eta(c)$ of E' will also be of the form $D[s_1, \dots, s_m](G_1, \dots, G_n)$ and the variational norm will be zero for $y_1 = G_1, \dots, y_n = G_n$. If the minimum exists, the optimal value G_j of the parameter c_j (identified by name) is added to the output database and to the environment used in the evaluation of the rest of the table. Otherwise, the global column name c is mapped to **static**(fail) in the output database and the local name x is mapped to **static**(fail) in the environment, meaning that all subsequent expressions depending on x will evaluate to fail. Note that the minimum of the variational norm may, in some degenerate cases, not be unique, so the semantics is technically not deterministic.

The rule (VAL QUERY INST) is similar, except that it computes the optimal value of c_j in each row separately. If the minimum does not exist in some row i , only the value $V_{i,j}$ in that row is set to fail, not all the values in the given column.

The rules (VAL QUERYORDET STATIC) and (VAL QUERYORDET INST) just evaluate expressions in non-**rnd** columns not containing **infer** operators deterministically, in the environments δ and τ .

Finally, we define the top-level judgment $\delta_{in}; \delta \vdash_{\sigma} \mathbb{S} \Downarrow \delta_{out}$ which states that in the schema-level environment δ and given the marginal map σ , the schema \mathbb{S} with database $DB = (\delta_{in}, \rho_{sz})$ evaluates to the map δ_{out} , storing the inferred values of all **qry** columns (as well as the values of deterministic columns).

Schema evaluation: $\delta_{in}; \delta \vdash_{\sigma} \mathbb{S} \Downarrow \delta_{out}$

(QUERY SCHEMA EMPTY)	(QUERY SCHEMA TABLE)
	$t; \delta_{in}; \delta; \emptyset; \sigma(t) \vdash \mathbb{T} \Downarrow \tau_t$
$\delta_{in}; \delta \vdash_{\sigma} [] \Downarrow []$	$\delta_{in}; \delta, (t \rightarrow \tau_t) \vdash_{\sigma} \mathbb{S} \Downarrow \delta_{out}$
	$\delta_{in}; \delta \vdash_{\sigma} (t = \mathbb{T}) :: \mathbb{S} \Downarrow (t \mapsto \tau_t) :: \delta_{out}$

The rule (QUERY SCHEMA EMPTY) is obvious, while (QUERY SCHEMA TABLE) evaluates the first table by calling the table-level evaluation judgment.

Example of Query Semantics To see how the query semantics computes the output database, let us revisit the simplified version of the Old Faithful example once again:

table faithful			
duration.Mean > Mean	real!rnd	static output	GaussianFromMeanAndPrecision(0.0, 1.0)
duration > duration	real!rnd	output	GaussianFromMeanAndPrecision(Mean, 1.0)
posteriorMean > postMean	real!qry	static output	infer.Gaussian.mean(Mean)

Again, we assume that the input database is $DB = (\delta_{in}, \rho_{sz})$, where $\delta_{in} = [\text{faithful} \mapsto [\text{duration.Mean} \mapsto \mathbf{static}(?), \text{duration} \mapsto \mathbf{inst}(4.0, 5.0, 4.0)]]$ and $\rho_{sz} = [\text{faithful} \mapsto 3]$. We have already shown that the marginal map $\sigma(\mathbb{S}, DB)$ for this model is $\sigma(\mathbb{S}, DB) = [(\text{faithful} \mapsto [\text{posteriorMean} \mapsto \mathbf{static}(G(3.25, 4))])]$, where G is the Gaussian distribution with mean and precision parameters. We want to find δ_{out} such that $\delta_{in}; \delta \vdash_{\sigma} \mathbb{S} \Downarrow \delta_{out}$, where $\sigma = \sigma(\mathbb{S}, DB)$.

To derive the above judgment by (QUERY SCHEMA TABLE), we need to first derive its assumption $\text{faithful}; \delta_{in}; \emptyset; \emptyset; \eta \vdash \mathbb{T} \Downarrow \tau_t$, where $\eta = [\text{posteriorMean} \mapsto \mathbf{static}(G(3.25, 4))]$ and $\delta_{out} = [\text{faithful} \mapsto \tau_t]$.

The first two columns, in space **rnd**, are discarded by the (VAL RANDOM) rule. Hence, we only need to derive $\text{faithful}; \delta_{in}; \emptyset; \emptyset; \eta \vdash \mathbb{T}' \Downarrow \tau_t$, where \mathbb{T}' contains only the last column.

The above judgment can be derived with (VAL QUERY STATIC):

(VAL QUERY STATIC)

$$(G_{\text{Mean}}, G_{\text{Var}}) = \arg \min_{y_1, y_2} \|\text{Gaussian}(y_1, y_2) - \eta(\text{posteriorMean})\|$$

$$\frac{\text{faithful}; \delta_{in}; \emptyset; [(\text{postMean} \rightarrow \mathbf{static}(G_{\text{Mean}}))]; \eta \vdash \square \Downarrow \square}{\text{faithful}; \delta_{in}; \emptyset; \emptyset; \eta \vdash}$$

$$[(\text{posteriorMean} \triangleright \text{postMean} : \mathbf{real} ! \mathbf{qry} \mathbf{static} \mathbf{output} (\mathbf{infer}.\text{Gaussian}.\text{mean}(\text{Mean})))]$$

$$\Downarrow [\text{posteriorMean} \mapsto \mathbf{static}(G_{\text{Mean}})]$$

We have $\eta(\text{posteriorMean}) = G(3.25, 4)$, so the values of y_1, y_2 minimising the variational norm (to zero) are $y_1 = 3.25$ and $y_2 = 0.25$ (note that are fitting the Gaussian parametrised by mean and variance to $G(3.25, 4)$, in which 4 is precision, the inverse of variance). Hence, $G_{\text{Mean}} = 3.25$, and so $\tau_t = [\text{posteriorMean} \mapsto \mathbf{static}(3.25)]$.

Therefore, by (QUERY SCHEMA TABLE), we get $\delta_{out} = [\text{faithful} \mapsto [\text{posteriorMean} \mapsto \mathbf{static}(3.25)]]$.

4.5.6 Output Database Conformance

Finally, we can formalise and prove the main result of this section, which states that the query semantics reduces each well-typed schema \mathbb{S} and a conformant input database DB , using the marginal map $\sigma(\mathbb{S}, DB)$, to a valid output map δ_{out} . We begin by formally defining the conformance of an output database to a schema type—this differs from the definition of input database conformance by the fact that we require all **output** columns to have well-typed, non-null values in the database and that we need to account for **local** values (which do not appear in table types).

We first define the relation $(\tau; \rho_{sz}) \models_n^{out} Q$, stating that the table-level output map τ ,

in which each **inst**-level column has n rows, together with the size map ρ_{sz} , conforms to the table type Q . This relation is defined to be the least relation closed under the rules shown below. We write $\text{value}(G)$ to mean that the generalised value G is in fact a value—that is, it is not `fail` and contains no `fail` as a subexpression in case it is an array.

Conformance Rules for Table-level Output Maps: $(\tau; \rho_{sz}) \models_n^{out} Q$

(CONF \square OUT)

$$\overline{(\square; \rho_{sz}) \models_n^{out} \square}$$

(CONF STATIC INPUT OUT) (where $\text{space}(T) \neq \text{rnd}$)

$$\emptyset \vdash^{\text{static}} G : \rho_{sz}(T) \quad \text{value}(G)$$

$$(\tau; \rho_{sz}) \models_n^{out} Q$$

$$\overline{((c \mapsto \text{static}(G)) :: \tau; \rho_{sz}) \models_n^{out} (c \triangleright x : T \text{ static input}) :: Q}$$

(CONF INST INPUT OUT) (where $\text{space}(T) \neq \text{rnd}$)

$$\emptyset \vdash^{\text{inst}} G_i : \rho_{sz}(T) \quad \forall i \in 0..n-1$$

$$\text{value}(G_i) \quad \forall i \in 0..n-1$$

$$(\tau; \rho_{sz}) \models_n^{out} Q$$

$$\overline{((c \mapsto \text{inst}([G_1, \dots, G_n])) :: \tau; \rho_{sz}) \models_n^{out} (c \triangleright x : T \text{ inst input}) :: Q}$$

(CONF STATIC OUTPUT OUT) (where $\text{space}(T) \neq \text{rnd}$)

$$\emptyset \vdash^{\text{static}} G : \rho_{sz}(T)$$

$$\text{det}(T) \Rightarrow \text{value}(G)$$

$$(\tau; \rho_{sz}) \models_n^{out} Q$$

$$\overline{((c \mapsto \text{static}(G)) :: \tau; \rho_{sz}) \models_n^{out} (c \triangleright x : T \text{ static output}) :: Q}$$

(CONF INST OUTPUT OUT) (where $\text{space}(T) \neq \text{rnd}$)

$$\emptyset \vdash^{\text{inst}} G_i : \rho_{sz}(T) \quad \forall i \in 0..n-1$$

$$\text{det}(T) \Rightarrow \text{value}(G_i) \quad \forall i \in 0..n-1$$

$$(\tau; \rho_{sz}) \models_n^{out} Q$$

$$\overline{((c \mapsto \text{inst}([G_1, \dots, G_n])) :: \tau; \rho_{sz}) \models_n^{out} (c \triangleright x : T \text{ inst output}) :: Q}$$

(CONF STATIC LOCAL OUT)

$$(\tau; \rho_{sz}) \models_n^{out} Q$$

$$\overline{((c \mapsto \text{static}(G)) :: \tau; \rho_{sz}) \models_n^{out} Q}$$

(CONF INST LOCAL OUT)

$$\frac{(\tau; \rho_{sz}) \models_n^{out} Q}{((c \mapsto \mathbf{inst}([G_1, \dots, G_n])) :: \tau; \rho_{sz}) \models_n^{out} Q}$$

(CONF SKIP OUT)

$$\frac{\mathbf{rnd}(T) \quad (\tau; \rho_{sz}) \models_n^{out} Q}{(\tau; \rho_{sz}) \models_n^{out} (c \triangleright x : T \ell \text{ viz}) :: Q}$$

The output database conformance rules are different from the input database conformance rules in that the map τ is constructed dynamically in the conclusions of the rules, rather than assumed to be constant. This reflects the fact that these maps are created by evaluating subsequent columns of the table and adding corresponding entries. The first rule (CONF \square OUT) states that an empty table-level database conforms to the empty table type. The (CONF STATIC INPUT OUT) and (CONF INST INPUT OUT) rules only make sure that the values of **input** columns (not marked as random) are correctly copied to the output database. The rules (CONF STATIC OUTPUT OUT) and (CONF INST OUTPUT OUT) state that if we append an **output** column to a table type, an entry with a well-typed generalised value (or array of values) must be added to a conformant output map τ for the resulting map to conform to the extended type. As inference may fail, this generalised value (or one or more of the components of the array of generalised values) may be `fail` if the given column is a **qry** column, but is expected to be a proper value if the column is deterministic. Since the output map τ contains also values for local columns, not present in the type Q , the rules (CONF STATIC LOCAL OUT) and (CONF INST LOCAL OUT) allow a well-formed map to contain entries for such columns (which are not checked in any way, as there is nothing they could conform to). Finally, (CONF SKIP OUT) discards random columns, which are not supposed to be present in the output database—they are parts of the probabilistic model, not parts of the query.

We complete the definition of output database conformance by defining the top-level conformance relation $(\delta; \rho_{sz}) \models^{out} Sty$, stating that $(\delta; \rho_{sz})$ is a valid output database for a schema with type Sty . This relation is defined to be the least relation closed under the following rules:

Conformance Rules for Output Databases: $(\delta; \rho_{sz}) \models^{out} Sty$

(CONF SCHEMA \square OUT)

$$\frac{}{(\square; \rho_{sz}) \models^{out} \square}$$

(CONF SCHEMA TABLE OUT)

$$\frac{\rho_{sz}(t) \in \mathbb{N} \quad (\tau; \rho_{sz}) \models_{\rho_{sz}(t)}^{out} Q \quad (\delta, \rho_{sz}) \models^{out} Sty}{((t \mapsto \tau) :: \delta; \rho_{sz}) \models^{out} (t : Q) :: Sty}$$

The rule (CONF SCHEMA \square OUT) says that an empty database conforms to the empty schema type, while (CONF SCHEMA TABLE OUT) states that if a database (δ, ρ_{sz}) conforms to the schema type \mathbb{S} , then this database with the entry $(t \mapsto \tau)$ added to δ conforms to $(t : Q) :: Sty$ if τ conforms to Q .

Proof of Output Database Conformance We can now state and prove the main theorem in this section, saying that the combination of the two semantics defined above, applied to a well-typed schema with a conformant input database, computes a well-typed output database.

Theorem 2 (Conformance) *If $(\delta_{in}, \rho_{sz}) \models Sty$, and $\text{Core}(\mathbb{S})$ and $\emptyset \vdash \mathbb{S} : Sty$ then there exists a δ_{out} such that $\delta_{in} \vdash \mathbb{S} \Downarrow_{\sigma} \delta_{out}$, where $\sigma = \sigma(\mathbb{S}, DB)$. Moreover, $(\delta, \rho_{sz}) \models_{out} Sty$.*

In order to prove this theorem, we first split it into three lemmas. The first one states that the function $\sigma(\mathbb{S}, DB)$, applied to a well-typed schema and a conformant database, produces a map of marginals conforming to the schema—that is, containing a measure or array of measures for every expression in an **infer** construct:

Marginal Map Conformance: $\eta \models_n^{marg} \mathbb{T}, (\sigma; \rho_{sz}) \models^{marg} \mathbb{S}$

(CONF MARG TABLE)

$$\eta(c) = \mathbf{static}(v)$$

$$\forall c \in \text{qry_cols_static}(\mathbb{T})$$

$$\eta(c) = \mathbf{inst}([v_0, \dots, v_{n-1}])$$

$$\forall c \in \text{qry_cols_inst}(\mathbb{T})$$

$$\eta \models_n^{marg} \mathbb{T}$$

(CONF MARG SCHEMA)

$$\frac{(\sigma(t); \rho_{sz}) \models_{\rho_{sz}(t)}^{marg} \mathbb{S}(t) \quad \forall t \in \text{dom}(\mathbb{S})}{(\sigma; \rho_{sz}) \models^{marg} \mathbb{S}}$$

Lemma 12 *If $\text{Core}(\mathbb{S})$ and $\Gamma \vdash \mathbb{S} : Sty$ and $(\delta_{in}; \rho_{sz}) \models Sty$, then $(\sigma(\mathbb{S}; (\rho_{sz}, \delta_{in})); \rho_{sz}) \models^{marg} \mathbb{S}$.*

Proof: Let $\sigma = \sigma(\mathbb{S}; (\rho_{sz}, \delta_{in}))$. Take any t in $\text{dom}(\mathbb{S})$ and let $\mathbb{T} = \mathbb{S}(t)$. Then take any column $\text{col} = (c \triangleright x : T \ell \text{ viz } M)$ of \mathbb{T} such that $M = \mathbf{infer}.D[e_1, \dots, e_m].c_j(E')$. Then, by the definition of $\sigma(\mathbb{S}; (\rho_{sz}, \delta_{in}))$, if $\ell = \mathbf{static}$, we have $\sigma(t)(c) = \mathbf{static}(\widehat{\text{marg}}((\mathbb{S}, DB), t, c))$. By Corollary 11, $\widehat{\text{marg}}((\mathbb{S}, DB), t, c) = v$, where v is a probability measure on $(\mathbb{R}, \mathcal{R})$.

Meanwhile, if $\ell = \mathbf{inst}$, we have $\sigma(t)(c) = \mathbf{inst}([\text{for } i < \rho_{sz}(t) \rightarrow \widehat{\text{marg}}((\mathbb{S}, DB), t, c, i)])$. By Corollary 11, $\widehat{\text{marg}}((\mathbb{S}, DB), t, c, i) = v_i$, for every $i \in 0.. \rho_{sz}(t) - 1$, where v_i is a probability measure on $(\mathbb{R}, \mathcal{R})$.

Hence, for every $t \in \text{dom}(\mathbb{S}(t))$, we have $(\sigma(t); \rho_{sz}) \models_{\rho_{sz}(t)}^{\text{marg}} \mathbb{S}(t)$ by (CONF MARG TABLE), and so $(\sigma; \rho_{sz}) \models^{\text{marg}} \mathbb{S}$ by (CONF MARG SCHEMA), as required. ■

Next, we show that applying the query semantics to a well-typed schema with a conformant database and a conformant map of marginals actually yields an output database. A detailed proof of this fact is deferred until the appendix.

Lemma 13 *If $\text{Core}(\mathbb{S})$ and $\emptyset \vdash \mathbb{S} : \text{Sty}$ and $(\delta_{in}, \rho_{sz}) \models \text{Sty}$ and $(\sigma, \rho_{sz}) \models^{\text{marg}} \mathbb{S}$ then $\delta_{in}, [] \vdash \mathbb{S} \Downarrow_{\sigma} \delta_{out}$ for some δ_{out} .*

Proof: In Appendix C ■

Finally, we show that every output database computed by the semantics for a well-typed schema with a conformant database conforms to the schema type. Again, the details of the proof are in the appendix.

Lemma 14 *If $\text{Core}(\mathbb{S})$ and $\emptyset \vdash \mathbb{S} : \text{Sty}$ and $(\delta_{in}, \rho_{sz}) \models \text{Sty}$ and $\delta_{in}, [] \vdash_{\sigma} \mathbb{S} \Downarrow \delta_{out}$, then $(\delta_{out}; \rho_{sz}) \models^{\text{out}} \text{Sty}$.*

Proof: In Appendix C ■

The main theorem is the corollary of the three above lemmas.

Restatement of Theorem 2 *If $(\delta_{in}, \rho_{sz}) \models \text{Sty}$, and $\text{Core}(\mathbb{S})$ and $\emptyset \vdash \mathbb{S} : \text{Sty}$ then there exists a δ_{out} such that $\delta_{in} \vdash \mathbb{S} \Downarrow_{\sigma} \delta_{out}$, where $\sigma = \sigma(\mathbb{S}, DB)$. Moreover, $(\delta, \rho_{sz}) \models_{\text{out}} \text{Sty}$.*

Proof: By Lemma 12, we have $(\sigma(\mathbb{S}; (\rho_{sz}, \delta_{in})); \rho_{sz}) \models^{\text{marg}} \mathbb{S}$. Now, let $\sigma = \sigma(\mathbb{S}; (\rho_{sz}, \delta_{in}))$. By Lemma 13, $\delta_{in}, [] \vdash \mathbb{S} \Downarrow_{\sigma} \delta_{out}$ for some δ_{out} . By Lemma 14, $(\delta_{out}; \rho_{sz}) \models^{\text{out}} \text{Sty}$. ■

4.6 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. We also defined the semantics of the language, as a combination of a random semantics computing weights of random traces and a pseudo-deterministic query semantics using integrals of these weights to compute the expected values of queries depending on parameters of inferred distributions.

Possible directions of future work include adding support for inference in time-series models and allowing nested inference by extending the lattice of binding times, so that the distributions computed in one run of inference could be queried by the probabilistic model “active” in the following run.

Individual Contributions

The type system and reduction relation for Tabular presented in [Gordon et al., 2015], were designed in a team effort, in collaboration with other authors. However, the updated syntax of Tabular using separate internal and external column names and the new versions of the type system and reduction relation presented here are all my own work. The proof of type soundness for Tabular, the new semantics of Tabular presented in Section 4.5 and the proofs of correctness of the semantics are also entirely my own effort.

Chapter 5

Fabular: Tabular with Regression Formulae

Acknowledgement This chapter is based on the paper “Fabular: Regression Formulas as Probabilistic Programming” [Borgström et al., 2016] published at the 2016 Symposium on Principles of Programming Languages (POPL). The paper was joint work with Johannes Borgström, Andrew D. Gordon, Long Ouyang, Claudio Russo and Adam Ścibior.

The Tabular language, presented in Chapter 4, aims to bring probabilistic programming to the large mass of casual users. However, the current statisticians’ weapon of choice is the R language, which features a simple yet useful and popular formula notation for expressing linear models with group-level coefficients, which are a version of linear models in which the values of regression coefficients may be different for data points with a different value of some categorical predictor. This domain-specific language, which can be used in conjunction with several inference packages such as `lm` and `lmer`, allows a very concise representation of a class of models frequently used in statistics, but lacks any formal semantics, with the meaning of formulae only defined by the implementations of inference packages and informal textual descriptions in the R documentation.

This chapter introduces Fabular, an extension of the Tabular language presented in Chapter 4, with hierarchical regressions. This language includes an improved version of the R formula language (supporting proper hierarchical regressions, whereby coefficients can themselves be modelled by other regressions), which can be used as ordinary model expressions in Tabular. By providing a translation from Fabular to Core Tabu-

lar, which has a formally defined semantics, we also give a rigorous semantics to the calculus. To ensure correctness of the embedding, we also define a type system of the regression calculus and show that Fabular schemas with well-typed formulas reduce to well-typed Core Tabular programs.

5.1 Linear Regression Formulae in R and Their Limitations

The goal of usual (univariate) linear regression is to fit a line of the form $y_i = \alpha \times x_i + \beta + \varepsilon_i$, where ε_i is the i -th error term, to a dataset consisting of points (x_i, y_i) . The unknown parameters α and β are called *slope* and *intercept*, respectively. Obviously, this form of regression generalises trivially to the multivariate case. In the basic form of linear regression with *group-level coefficients* (sometimes called *random effects*), we assume that we additionally have a discrete *categorical* predictor c , admitting a possibly different value c_i in each i -th row, and that we have multiple values of either α or β , one for each possible value of c . For instance, consider the formula $y_i = \alpha[c_i] \times x_i + \beta + \varepsilon_i$, where each c_i is an integer in the range $[0, n-1]$, and assume that each row of data consists of values (x_i, y_i, c_i) . In this regression, α is considered to be an array of n parameters, and in each row i , the component of α to be used is determined by the value c_i of c in this row.

In the R formula language, the parameter names are anonymised, the intercept and error term are added automatically and the dependence of the parameter of a variable x on a categorical predictor c is denoted by $x|c$. Hence, the above hierarchical regression formula can be written simply as $y \sim x|c$.

The main package for linear modelling with group-level coefficients, *lmer*, only performs non-Bayesian maximum likelihood computation and does not allow defining priors on coefficients. The *blme* package [Dorie, 2016] added the possibility to define priors on global and group-level regression coefficients, allowing for defining *hierarchical linear models*, whereby coefficients of a given linear regression themselves have probabilistic models. However, the priors have to be specified outside the formula language, and they are limited to Gaussian and Student t distributions in case of global coefficients—meanwhile, for group-level coefficients, one can only define a prior on their covariance matrix, which may complicate modelling.

Motivated in equal measure by the popularity and the shortcomings of the R for-

mula language, we define a compositional calculus for hierarchical linear regressions, in which global and group-level coefficients can themselves be modelled just like top-level regressions, by the use of recursive syntax. This conforms to the spirit of hierarchical regression and allows defining coefficients with arbitrary priors, or even deeper models where the coefficients in the model of a top-level coefficient are themselves modelled by higher-level regressions.

We present a type system and semantics of this regression calculus, which subsumes the R formula language, and embed the calculus in Tabular, showing how formulae can be used as a convenient domain-specific language within a more standard probabilistic programming system, shortening and simplifying larger probabilistic models.

5.2 Syntax of the Regression Calculus

The syntax of the calculus consists of *predictors* and *regressions*. Because of the compositionality of the calculus, regressions can model group-level coefficients dependent on multiple categorical variables, which admit different values for different combinations of these grouping variables. Such coefficients can be treated as multidimensional arrays, with one dimension corresponding to each grouping factor. Hence, the predictors and regressions, despite seemingly returning scalar values, define in fact multidimensional arrays (of dimension zero in the case of top-level scalar expressions).

To reason about predictors and regressions in nested, hierarchical models, without referencing their dimensionality directly, we introduce *cube-expressions*, which represent arrays of arbitrary dimensionality. We write \vec{e} for the *dimension* of a cube, being a finite list $[e_0, \dots, e_{n-1}]$ of non-negative integers. A cube-expression of base type T and dimension \vec{e} is a multi-dimensional array of type $T[e_{n-1}] \dots [e_0]$. An *index* to a cube of size $\vec{e} = [e_0, \dots, e_{n-1}]$ is a list $\vec{i} = [i_0, \dots, i_{n-1}]$ of integers such that $0 \leq i_j < e_j$ for each $j \in 0..n-1$. If E is a n -dimensional array and $\vec{i} = [i_0, \dots, i_{n-1}]$, we write $E[\vec{i}]$ for $E[i_{n-1}] \dots [i_0]$.

Predictors are deterministic quantities representing the data used in the model. Their syntax is as follows:

Predictors:

$u, v ::=$	predictor
s	scalar (typically 0 or 1)
x	variable (discrete or continuous)

$u : v$	interaction
$(u_1, \dots, u_n).v$	path expression

A *scalar* s is a cube whose every element is s , which is typically 1, denoting the presence of an intercept term. A *variable* x simply denotes the cube corresponding to x . An *interaction* $u : v$ is the pointwise product of the cubes defined by u and v . Finally, a path expression $(u_1, \dots, u_n).v$ denotes the cube defined by v composed with the *index transformers* defined by u_1, \dots, u_n —that is, if v defines a cube E and u_1, \dots, u_n are cubes E_1, \dots, E_n , then the cube $(u_1, \dots, u_n).v$ applied to the index $\vec{j} = [j_1, \dots, j_m]$ returns $E[E_1[\vec{j}]] \dots [E_n[\vec{j}]]$. The purpose of path expression is to reference a variable which has the same dimensionality as the parameters, and not outputs, of a given expression. Its main application is in defining priors on noise terms, as described later in this section, in which case the parameter list u_1, \dots, u_n is empty and the predictor v is expected to be a real-valued variable.

Regressions have the following syntax:

Regressions:

$r ::=$	regression
$D[e_1, \dots, e_m](v_1, \dots, v_n)$	noise
$v\{\alpha \sim r\}$	modelled predictor
$r + r'$	sum
$r v$	grouping
$x \sim r$ in r'	local binding (x alpha-convertible)

The *noise* $D[e_1, \dots, e_m](v_1, \dots, v_n)$ can be drawn from an arbitrary real-valued distribution D . We reuse the syntax and signatures of continuous distributions from Tabular (as defined in Chapter 4), except that we add the Dirac delta distribution (putting all the probability mass on a single point) and an alternative parametrisation of Gamma, with scale replaced by rate (inverse of scale).

The list of distributions supported is the following (but could easily be extended):

Real-Valued Distributions: $D_{spc} : [x_1 : T_1, \dots, x_m : T_m](y_1 : U_1, \dots, y_n : U_n) \rightarrow \mathbf{real} ! \mathbf{rnd}$

$\text{Beta}_{spc} :: (a : \mathbf{real}!spc, b : \mathbf{real}!spc) \rightarrow \mathbf{real} ! \mathbf{rnd}$

$\text{Gamma}_{spc} : (\text{shape} : \mathbf{real}!spc, \text{scale} : \mathbf{real}!spc) \rightarrow \mathbf{real} ! \mathbf{rnd}$

$\text{Gaussian}_{spc} : (\text{mean} : \mathbf{real!spc}, \text{variance} : \mathbf{real!spc}) \rightarrow \mathbf{real!rnd}$
 $\text{GaussianFromMeanAndPrecision}_{spc} : (\text{mean} : \mathbf{real!spc}, \text{prec} : \mathbf{real!spc}) \rightarrow \mathbf{real!rnd}$
 $\text{GammaFromShapeAndRate}_{spc} : (\text{shape} : \mathbf{real!spc}, \text{rate} : \mathbf{real!spc}) \rightarrow \mathbf{real!rnd}$
 $\text{Dirac}_{spc} : (\text{point} : \mathbf{real!spc}) \rightarrow \mathbf{real!rnd}$

From the above list, the Gaussian and Gamma distributions are most commonly used in linear models.

The modelled predictor $v\{\alpha \sim r\}$ denotes a predictor whose parameter has a prior defined by r . The predictor v can be an arbitrary dimensional cube, and the parameter α can also be multi-dimensional, depending on the context. The sum $r + r'$ returns the pointwise sum of the cubes defined by regressions r and r' . The grouped regression $r|v$ is regression r with the top-level parameters turned into arrays indexed by v (or with one dimension added to the parameters if they are already arrays). Finally, the operator $x \sim r$ in r' binds the local variable x to the cube defined by r in the regression r' . The variable x is akin to a local parameter, rather than a modelled predictor, and its dimensionality must match the dimensionality of parameters in r' (as long as r' has no grouping factors).

When showing regression formulas, we assume that the operator $|$ binds more tightly than $+$.

Scoping Rules, Alpha-Conversion and Free Variables Regression parameters α are assumed to be fixed and unique; they are not α -convertible. Parameters are not binders and cannot be referenced anywhere in the given regression.

On the other hand, local variables x , introduced by the construct $x \sim r$ in r' , are α -convertible and can be referenced in the regression: in $x \sim r$ in r' , the variable x is bound in r' . This is to allow, for instance, defining the prior of a parameter of a noise term by another linear regression.

The set of free variables of a predictor or regression is defined below. Note that parameters are not free variables.

Free Variables in Predictors and Regressions: $\text{fv}(v), \text{fv}(r)$

$$\text{fv}(s) = \emptyset$$

$$\text{fv}(x) = \{x\}$$

$$\text{fv}(u : v) = \text{fv}(u) \cup \text{fv}(v)$$

$$\text{fv}((u_1, \dots, u_n).v) = \text{fv}(u_1) \cup \dots \cup \text{fv}(u_n) \cup \text{fv}(v)$$

$$\begin{aligned}
\text{fv}(D[k_1, \dots, k_m](u_1, \dots, u_n)) &= \text{fv}(u_1) \cup \dots \cup \text{fv}(u_n) \\
\text{fv}(v\{\alpha \sim r\}) &= \text{fv}(v) \cup \text{fv}(r) \\
\text{fv}(r_1 + r_2) &= \text{fv}(r_1) \cup \text{fv}(r_2) \\
\text{fv}(r|v) &= \text{fv}(r) \cup \text{fv}(v) \\
\text{fv}(x \sim r \text{ in } r') &= \text{fv}(r) \cup (\text{fv}(r') \setminus \{x\})
\end{aligned}$$

Abbreviations As the extended and more flexible modelling syntax arguably makes the regression calculus wordier than the R formula language which inspired it, we introduce several abbreviations allowing to write common models with default priors more concisely. The term $v\{\alpha\}$ denotes a predictor v with parameter r whose prior is a normal distribution with large variance, which is meant to simulate an uninformative prior.

$$v\{\alpha\} = v\{\alpha \sim \text{Gaussian}(0, s_{large}^2)\}$$

The noise term will typically be a draw from the Gaussian distribution. We let $? \{\sim r\}$ be the Gaussian noise with the prior on precision defined by the regression r :

$$? \{\sim r\} = x \sim r \text{ in } \text{GaussianFromMeanAndPrecision}(0, () .x)$$

Note that x is a scalar variable, not a cube, so to access it we need to use the path expression $() .x$ with empty path. If we referenced it as just x , it would be incorrectly treated as a predictor having the same dimensionality as the output cube of the noise term itself.

The term $?$ stands for Gaussian noise whose precision has a Gamma prior with a large rate.

$$? = ? \{\sim \text{GammaFromShapeAndRate}(0, s_{large})\}$$

Example: Cheese Sales To demonstrate the concision of the regression calculus, let us now consider an example based on the `cheese` dataset from R's `bayesm` package (<https://www.rdocumentation.org/packages/bayesm>). Suppose we have a database storing the price of a pack of sliced cheese and the number of packs sold in some year in stores belonging to various retail chains in different American cities.

Cities		Chains	
ID	city	ID	chain
0	LOS ANGELES	0	LUCKY
1	CHICAGO	1	RALPHS
2	HOUSTON	2	KROGER CO
(...)		(...)	

Sales				
ID	city	chain	price	volume
0	LOS ANGELES	LUCKY	2.57846	21374
1	LOS ANGELES	RALPHS	3.727867	6427
2	LOS ANGELES	VONS	2.711421	17302
3	CHICAGO	DOMINICK	2.651206	13561
4	CHICAGO	JEWEL	1.986674	42774
5	CHICAGO	OMNI	2.386616	4498
6	HOUSTON	KROGER CO	2.481124	6834
7	HOUSTON	RANDALLS	3.428268	3764
8	DETROIT	KROGER CO	2.747321	5505
9	SAN FRANCISCO	LUCKY	3.716438	6041
(...)				

Like in Tabular, we assume that each table has a numeric primary key ID and that links are integers, but for the sake of readability we add additional string columns to the tables Cities and Chains and use them as aliases for corresponding indices (so, for example, HOUSTON is an alias for the index 2).

We are interested in checking how the sales volume depends on the price of this cheese. In the simplest possible model, we may assume that the sales volume does not depend on the price:

$$\text{volume} \sim 1\{\alpha\} + ?$$

In this model, we assume that the volume is always a noisy copy of a single, global intercept α , drawn from a Gaussian with a large variance. As it is not realistic to expect the sales not to depend on the price, we can add another term to the above regression, to make it a linear function of the price:

$$\text{volume} \sim 1\{\alpha\} + \text{price}\{\beta\} + ?$$

Now, the regression contains an *intercept* α , drawn from a normal distribution, and a *slope* term $\text{price}\{\beta\}$, being the price multiplied by the proportionality rate β , also sampled from a wide normal distribution (and, obviously, expected to be negative). Again, we need to add the noise term ϵ , as we cannot require all the observations to match the model exactly.

The above model does not take into account cities and chains—the inferred parameters α and β are the same for every row of the Sales table. However, we might expect that, for instance, how much price affects sales will depend on the retailer, as customers of more upmarket stores are likely to pay less attention to the price. To account for this, we can modify the model as follows:

$$\text{volume} \sim 1\{\alpha\} + \text{price}\{\beta\}|\text{chain} + \epsilon$$

Now, we still have one global value of the intercept α , but the slope β is allowed to be different in entries concerning different chains. In other words, we have one value of β per chain.

Finally, we can expect the intercept of regression, representing the baseline demand for cheese, to depend on the city, as the demand should be higher in more populous areas. We can adapt the model as follows:

$$\text{volume} \sim 1\{\alpha\}|\text{city} + \text{price}\{\beta\}|\text{chain} + \epsilon$$

We now have one value of α per city and one value of β per chain.

Example: Radon measurements and partial pooling In the above model, all the regression parameters have default Gaussian priors, and so the nested regressions modelling parameters are simple Gaussian distributions with no input variables.

To show the flexibility of the regression calculus, let us consider a more complex model adapted from [Gelman and Hill, 2007], modelling radon radiation levels in houses in different counties of Minnesota. The dataset for this model consists of (continuous) measurements of radon activity in houses in different counties, together with the floor on which the given measurement was taken (which can be treated as a continuous predictor). Additionally, each county has an associated uranium level.

Counties			Measurements			
ID	county	uranium	ID	county	floor	activity
0	AITKIN	0.502054	0	AITKIN	1	2.2
1	ANOKA	0.428565	1	AITKIN	0	2.9
2	BECKER	0.892741	2	ANOKA	0	1.7
(...)			3	BECKER	1	1.2
			(...)			

Suppose that we expect the radon activity in each house to depend on the level of uranium radiation in the given county and the floor on which the measurement was taken (as the uranium radiation coming from the ground should be weaker on the upper floor). We can then model radon activity by the following regression:

$$\text{activity} \sim 1\{\alpha \sim r\}|\text{county} + \text{floor}\{\beta\} + ?$$

where:

$$r = 1\{\gamma\} + \text{uranium}\{\zeta\} + ?$$

In this model, the radon activity in each house is a linear combination of a county-level intercept α , modelled by the regression r and admitting different values for houses in different counties, the floor on which the measurement was taken multiplied by the global proportionality constant β and a default noise term $?$. The regression r consists of a global intercept γ and the uranium level in the given county multiplied by some parameter ζ , having a default Gaussian prior. Note that while in the top-level regression, both the input floor and the modelled output activity are in the top-level table **Measurements**, the input uranium of the nested regression r is in the table **Counties**, and the regression also produces one output per entry of this table. By conditioning r on the predictor county, which is a link to the table **Counties**, we specify that r has one input and output per county, not per measurement.

5.3 Typing Regression Formulae

We present a simple type system for regression formulae, which ensures that the given formula is well-formed and gives the list of parameters defined by a hierarchical regression, together with their types and dimensionalities. In the typing rules and judgments, we reuse the syntax of Tabular environments Γ , base types T and the typing judgment

for variables, $\Gamma \vdash^{pc} x : T$. When typechecking a regression r , we assume that all predictors occurring in r are defined in the *initial* environment Γ , corresponding to the database storing the values of predictors. These predictors are all expected to be **static** arrays, except for top-level predictors (i.e. not appearing in nested regressions), which can be at **inst** level. If a top-level predictor is an **inst**-level scalar, it actually means that it defines an array of the length equal to the size of the main table in the database, just like **inst**-level columns in Tabular.

We begin by giving the typing rules for predictors, which define the judgment $\Gamma; \vec{e}; \ell \vdash v : T$, which says that in the environment Γ , the predictor v defines a cube of dimensionality \vec{e} with base type T , at level ℓ (which can be **inst** only if the predictor is top-level). We write $T[\vec{e}]$ for $T[e_n] \dots [e_1]$ if $\vec{e} = [e_1, \dots, e_n]$. Formally, the elements of \vec{e} are Tabular *indexed expressions*, as defined in Section 4.2.2, so they can be scalars, variables present in the environment or table sizes. The type system requires these elements to be deterministic integers.

Typing rules for predictors: $\Gamma; \vec{e}; \ell \vdash v : T$

(SCALAR) (where $\vec{e} = [e_1, \dots, e_n]$)
$\Gamma \vdash \diamond \quad s \in \mathbb{R} \quad \Gamma \vdash^{\text{static}} e_i : \text{int}! \text{det} \quad \forall i \in 1..n$
$\Gamma; \vec{e}; \ell \vdash s : \text{real}! \text{rnd}$
(VAR)
$\Gamma \vdash^\ell x : T[\vec{e}]$
$\Gamma; \vec{e}; \ell \vdash x : T$
(INTERACT)
$\Gamma; \vec{e}; \ell \vdash u : \text{real}! \text{rnd} \quad \Gamma; \vec{e}; \ell \vdash v : \text{real}! \text{rnd}$
$\Gamma; \vec{e}; \ell \vdash u : v : \text{real}! \text{rnd}$
(PATH) (where $\vec{f} = [f_1, \dots, f_n]$)
$\Gamma; \vec{e}; \ell \vdash u_i : \text{mod}(f_i)! \text{det} \quad \forall i \in 1..n \quad \Gamma; \vec{f}; \ell \vdash v : T$
$\Gamma; \vec{e}; \ell \vdash (u_1, \dots, u_n).v : T$

The (SCALAR) rule only allows real-valued constants to be used in the calculus (as mentioned before, they are usually 1). The (VAR) rule states that if the variable x is supposed to represent a cube of dimension \vec{e} and base type T , it must have type $T[\vec{e}]$ in the environment. The following rule, (INTERACT), states that two predictors can only be multiplied if their dimensions match. Meanwhile, (PATH) states that in a path

expression, the elements u_i of a path must be cubes of bounded integers of matching dimensions and the predictor v must be a cube of dimensions specified by the bounds on path predictors. As stated before, the most common use case of the path expression is defining priors on noise terms, in which case the path u_1, \dots, u_n is empty and v is a scalar real-valued variable x . In this case the (PATH) rule simplifies to the following:

(PATH)

$$\frac{\Gamma \vdash^\ell x : \mathbf{real} ! \mathbf{rnd}}{\Gamma; \vec{e}; \ell \vdash () . x : \mathbf{real} ! \mathbf{rnd}}$$

The regression typing rules derive the judgment $\Gamma; \vec{e}; \vec{f}; \ell \vdash r ! \Pi$, which states that in the environment Γ , if the level of regression is ℓ (which must be **static** for nested regressions), the dimensionality of the modelled predictor is \vec{e} and the parameter dimensionality at the current level is \vec{f} , the regression r defines the list of parameters Π (which has the same syntax as an environment Γ). Elements of \vec{f} , just like elements of \vec{e} , are indexed expressions.

Typing rules for regressions: $\Gamma; \vec{e}; \vec{f}; \ell \vdash r ! \Pi$

$$\begin{array}{l} \text{(NOISE) (where } \vec{e} = [e_1, \dots, e_n] \text{ and } \vec{f} = [f_1, \dots, f_{n'}] \text{ and } \sigma(U) \triangleq U\{\hat{e}_1/x_1\} \dots \{\hat{e}_{m'}/x_{m'}\}) \\ D_{rnd} : [x_1 : T_1, \dots, x_{m'} : T_{m'}](c_1 : U_1, \dots, c_m : U_m) \rightarrow \mathbf{real} ! \mathbf{rnd} \\ \Gamma \vdash \diamond \quad \Gamma \vdash^{\mathbf{static}} e_i : \mathbf{int} ! \mathbf{det} \quad \forall i \in 1..n \quad \Gamma \vdash^{\mathbf{static}} f_i : \mathbf{int} ! \mathbf{det} \quad \forall i \in 1..n' \\ \Gamma \vdash^{\mathbf{static}} \hat{e}_i : T_i \quad \forall i \in 1..m' \quad \Gamma; \vec{e}; \ell \vdash u_j : \sigma(U_j) \quad \forall j \in 1..m \\ \{x_1, \dots, x_{m'}\} \cap (\bigcup_i \text{fv}(\hat{e}_i)) = \emptyset \quad x_i \neq x_j \text{ for } i \neq j \\ \hline \Gamma; \vec{e}; \vec{f}; \ell \vdash D[\hat{e}_1, \dots, \hat{e}_{m'}](u_1, \dots, u_m) ! \emptyset \end{array}$$

(COEFF)

$$\frac{\Gamma; \vec{e}; \ell \vdash v : \mathbf{real} ! \mathbf{rnd} \quad \Gamma; \vec{f}; []; \mathbf{static} \vdash r ! \Pi}{\Gamma; \vec{e}; \vec{f}; \ell \vdash v\{\alpha \sim r\} ! \Pi, \alpha : (\mathbf{real} ! \mathbf{rnd})[\vec{f}]}$$

(SUM)

$$\frac{\Gamma; \vec{e}; \vec{f}; \ell \vdash r ! \Pi \quad \Gamma; \vec{e}; \vec{f}; \ell \vdash r' ! \Pi'}{\Gamma; \vec{e}; \vec{f}; \ell \vdash r + r' ! \Pi, \Pi'}$$

(GROUP)

$$\frac{\Gamma; \vec{e}; \ell \vdash v : \mathbf{mod}(f) ! \mathbf{det} \quad \Gamma; \vec{e}; (f :: \vec{f}); \ell \vdash r ! \Pi}{\Gamma; \vec{e}; \vec{f}; \ell \vdash r|v ! \Pi}$$

(BIND)

$$\frac{\Gamma; \vec{f}; [], \mathbf{static} \vdash r ! \Pi \quad \Gamma, x : \mathbf{static} (\mathbf{real} ! \mathbf{rnd})[\vec{f}]; \vec{e}; \vec{f}; \ell \vdash r' ! \Pi'}{\Gamma; \vec{e}; \vec{f}; \ell \vdash x \sim r \text{ in } r' ! \Pi, \Pi'}$$

The (NOISE) rule says that the noise term itself does not add any parameters to the model, and requires that all predictors used as parameters of D have the right types. Like in Tabular, types U_j of parameter predictors may depend on hyperparameters \hat{e}_i , which are expected to be constants, table sizes or deterministic variables (which must be present in the initial environment, as there are no rules adding deterministic variables to the environment in the Fabular type system). The (COEFF) rule checks the type and dimensionality of the predictor v and recursively typechecks the nested regression r . As this nested regression models the parameter α of the top-level regression, the parameter dimensionality of the top-level regression becomes the modelled predictor dimensionality of r and the parameter dimensionality of r is initially empty. Meanwhile, (SUM) says that the list of parameters defined by a sum $r + r'$ of regressions is the concatenation of their individual parameter lists. The (GROUP) rule extends the current parameter dimension vector \vec{f} with the bound f of the categorical predictor v and checks the nested regression r with this extended vector $f :: \vec{f}$. This rule says that if $\vec{f} = []$ and r is a modelled regression of the form $v\{\alpha \sim r'\}$, the grouping factor effectively turns the parameter α into an array of size f . Finally, (BIND) checks that r defines a cube whose dimensionality matches the current parameter dimensionality \vec{f} and that the regression r' is well typed in the environment extended with x , denoting the cube defined by r .

Example Recall the cheese sales example from Section 5.2. By desugaring the last discussed regression for this dataset, $r = 1\{\alpha\}|\text{city} + \text{price}\{\beta\}|\text{chain} + ?$, we get

$$r = 1\{\alpha \sim \text{Gaussian}(0, s_{\text{large}}^2)\}|\text{city} + \text{price}\{\beta \sim \text{Gaussian}(0, s_{\text{large}}^2)\}|\text{chain} + \\ (x \sim \text{GammaSR}(1, s_{\text{large}}) \text{ in } \text{GaussianMP}(0, ().x))$$

(writing GaussianMP for GaussianFromMeanAndPrecision and GammaSR for GammaFromShapeAndRate).

Suppose the tables **Cities** and **Chains** have pre-determined sizes n_{Cities} and n_{Chains} , respectively. Then, an initial environment corresponding to the database can be of the form:

$$\Gamma = \text{city} :^{\text{inst}} \text{mod}(n_{\text{Cities}}) ! \text{det}, \text{chain} :^{\text{inst}} \text{mod}(n_{\text{Chains}}) ! \text{det}, \text{price} :^{\text{inst}} \text{real} ! \text{det},$$

Hence, it is easy to check that the type of this regression r in Γ is as follows:

$$\Gamma; [], []; \text{inst} \vdash r : \alpha : (\text{real} ! \text{rnd})[n_{\text{Cities}}], \beta : (\text{real} ! \text{rnd})[n_{\text{Chains}}]$$

Consider now the formula from the radon example:

$$\text{activity} \sim 1\{\alpha \sim r\}|\text{county} + \text{floor}\{\beta\} + ?$$

$$r = 1\{\gamma\} + \text{uranium}\{\zeta\} + ?$$

After desugaring, the regression r' modelling activity has the following form:

$$\begin{aligned} r' = & 1\{\alpha \sim r\}|\text{county} + \text{floor}\{\beta \sim \text{Gaussian}(0, s_{\text{large}}^2)\} + \\ & (x \sim \text{GammaSR}(1, s_{\text{large}}) \text{ in } \text{GaussianMP}(0, ().x)) \end{aligned}$$

where:

$$\begin{aligned} r = & 1\{\gamma \sim \text{Gaussian}(0, s_{\text{large}}^2)\} + \text{uranium}\{\zeta \sim \text{Gaussian}(0, s_{\text{large}}^2)\} \\ & (x \sim \text{GammaSR}(1, s_{\text{large}}) \text{ in } \text{GaussianMP}(0, ().x)) \end{aligned}$$

Let us assume that the table **Counties** has n_{Counties} rows. Then the environment matching the database has the following form:

$$\Gamma = \text{uranium} : \mathbf{static} (\mathbf{real} ! \mathbf{det})[n_{\text{Counties}}], \text{county} : \mathbf{inst} \mathbf{mod}(n_{\text{Counties}}) ! \mathbf{det}, \text{floor} : \mathbf{inst} \mathbf{real} ! \mathbf{det},$$

To derive the type of regression r' , by the rule (COEFF), we first need to typecheck the regression r at level **static**, with the output dimensionality set to $[n_{\text{Counties}}]$ (i.e. the parameter dimensionality of the term $1\{\alpha \sim r\}$, after setting \vec{f} to $[n_{\text{Counties}}]$ by (GROUP)). We can show that the type of r is as follows:

$$\Gamma; [n_{\text{Counties}}]; []; \mathbf{static} \vdash r : (\gamma : (\mathbf{real} ! \mathbf{rnd}), \zeta : (\mathbf{real} ! \mathbf{rnd}))$$

Hence, the type of the full regression r' is as follows:

$$\Gamma; []; []; \mathbf{inst} \vdash r' : (\gamma : (\mathbf{real} ! \mathbf{rnd}), \zeta : (\mathbf{real} ! \mathbf{rnd}), \alpha : (\mathbf{real} ! \mathbf{rnd})[n_{\text{Counties}}], \beta : (\mathbf{real} ! \mathbf{rnd}))$$

5.4 Fabular = Tabular + Regression Formulae

We now demonstrate how the domain-specific language of hierarchical linear regressions can be combined with a general probabilistic programming system by embedding the regression calculus in Tabular. We call the resulting language Fabular (Tabular + Formulas). We illustrate the succinctness and flexibility of the resulting language by several examples and provide a translation from Fabular to Core Tabular, just like for other compound Tabular models. Since the Core Tabular semantics is restricted in that

expressions in all conditioned **output** columns must be random draws, we require the regressions in Fabular to be of the form $r+?$, to ensure that the output expression of the regression can be represented as a draw from a Gaussian (centred at the output expression of r).

5.4.1 Syntax and Type System of Fabular

Fabular is an extension of Tabular with regression formulas, which are just another kind of compound model, just like function applications and indexing.

Full Fabular Schemas:

$M, N ::= \dots \mid \sim r+?$	model expression
--------------------------------	------------------

By adapting the typing judgment for formulae and translating parameter lists Π to Tabular Q -types, we can easily extend the Tabular type system to Fabular. We start by defining the following translation of regression types to Tabular types.

Translation of Parameter Lists to Tabular Model Types: $\llbracket \Pi \rrbracket$

$\llbracket \Pi, \alpha : T \rrbracket = \llbracket \Pi \rrbracket @ [(\alpha \triangleright y : T \text{ static output})]$ $\llbracket [] \rrbracket = \emptyset$

The local names y are irrelevant, because in regression types Π , parameters cannot be referenced in subsequent parameter types T . The level of all parameters is **static**, because parameters will be translated to arrays in **static** columns in the table where the regression was used.

With this type translation in place, we can define the additional typing rule for regression formulae.

Typing rules for Fabular tables: $\Gamma \vdash^{pc} M : Q$

(MODEL REGRESSION)
$\Gamma; \emptyset; []; []; \ell \vdash r+? ! \Pi$
$\Gamma \vdash^\ell r+? : \llbracket \Pi \rrbracket @ [(\text{ret} \triangleright y : \text{real} ! \text{rnd } \ell \text{ output})]$

This rule states that an embedded regression $r+?$ is only well-typed in Fabular if it is well-typed according to the type system for the calculus, and that the type of regression in Fabular is its list of parameters Π concatenated with the type of the top-level output expression defined by the regression. As in functions, we use the `ret` keyword to denote the output of the model. The level ℓ is the level of the column where the regression was used, and is normally expected to be **inst**—otherwise the regression could only model a single data point.

5.4.2 Translation to Core Tabular

We now show how Fabular programs can be reduced to Core Tabular, by reducing regression formulas to sequences of Tabular columns. We begin by showing the straight-forward rules for translating predictors to Tabular expressions. As mentioned before, a predictor will typically denote a multidimensional array, from which we will need to extract the right component when translating regressions. If $\vec{E} = [E_1, \dots, E_n]$ is a list of indices and v is a predictor of dimensionality $\vec{e} = [e_1, \dots, e_n]$ and each e_i is the bound of E_i , then $\llbracket v \rrbracket \vec{E}$ is the element of the cube defined by v at indices \vec{E} . Below, we write $x[\vec{E}]$ for $x[E_1] \dots [E_n]$.

Translation of predictors to Tabular expressions

$$\begin{aligned} \llbracket s \rrbracket \vec{E} &= s \\ \llbracket x \rrbracket \vec{E} &= x[\vec{E}] \\ \llbracket u : v \rrbracket \vec{E} &= \llbracket u \rrbracket \vec{E} \times \llbracket v \rrbracket \vec{E} \\ \llbracket (u_1, \dots, u_m).v \rrbracket \vec{E} &= \llbracket v \rrbracket [\llbracket u_1 \rrbracket \vec{E}] \dots [\llbracket u_m \rrbracket \vec{E}] \end{aligned}$$

The first case is obvious, as scalar predictors define cubes in which all entries are set to the given scalar. In the second case, the element of a cube defined by a variable is accessed simply by an ordinary array access in Tabular. In case of an interaction $u : v$, we need to get the elements of cubes defined by u and v at index \vec{E} , and multiply the results. In a path expression, the categorical predictors u_1, \dots, u_m define cubes from which we need to extract elements at index \vec{E} , and pass them as indices to the cube defined by v .

The translation of regressions is defined by means of the recursive function $\llbracket \ell; \vec{e}; \vec{f}; \vec{v}; r; K \rrbracket^\dagger$, where ℓ is the current level, r is the currently reduced regression, \vec{e} and \vec{f} are, respectively, output and parameter dimensionalities of r , \vec{v} is the list of

categorical grouping factors for r and $K = \lambda E.\mathbb{T}$ is a *continuation*, mapping a simple Tabular expression E to a Core table \mathbb{T} , possibly including the expression E (note that continuations are meta-language functions). This operator returns a table \mathbb{T} , which is a Tabular translation of the given regression r .

Before defining this operator, we present an additional model reduction rule reducing regressions to Tabular models. In order to reduce the regression as a model M , without looking at the table where it was used, and avoid having to replace references to parameters from outside the regression with corresponding, newly-introduced local variables (like in application reduction), we reduce a regression to a *function application*, applying the table returned by the aforementioned operator to an empty argument list. This way, references to the parameters in the outer table will be replaced by local variables in the next phase, when this dummy application is reduced.

Reducing Fabular to Core Tabular $M \rightarrow M'$

(RED REGR)

$$\frac{x \neq y \neq z}{\sim r + ? \rightarrow [\text{inst}; \square; \square; \square; r; K]^\dagger \square}$$

where

$$\begin{aligned} K &= \lambda E.(_ \triangleright x : \text{real} ! \text{rnd inst local } E) \\ &:: (_ \triangleright y : \text{real} ! \text{rnd static local } \text{GammaFromShapeAndRate}(0, s_{\text{large}})) \\ &:: (\text{ret} \triangleright z : \text{real} ! \text{rnd inst output } \text{GaussianFromMeanAndPrecision}(x, y)) \end{aligned}$$

In the above rule, the initial level can be set to **inst**, because if the regression was in a **static** column, all the columns would be reduced to **static** level when reducing the dummy function application. The initial continuation maps the output expression E of the regression r , to be computed by the translation operator, to a function table whose last **ret** column is a draw from a Gaussian centred at the value of the expression E , with a default Gamma prior. We set the expression in **ret** to a Gaussian centred at the value of E , rather than the sum of E and a Gaussian centred at 0, because the semantics of Fabular is only defined for schemas where the expressions in all conditioned columns are simple random draws. The auxiliary first column, binding x to E , is introduced to keep the invariant that the translation only uses continuations of the form $K = \lambda E.(c \triangleright x : T \ell \text{ viz } E) :: \mathbb{T}$ where \mathbb{T} does not depend on E , which will be useful in the proof of correctness of the translation.

The aforementioned operator $\llbracket \ell; \vec{e}; \vec{f}; \vec{v}; r; K \rrbracket^\dagger$, which performs the actual translation, is formally defined below. This translation is, strictly speaking, type-directed, as we need to know the bounds f of categorical predictors v . If $\vec{e} = [e_1, \dots, e_n]$ and $\vec{z} = [z_1, \dots, z_n]$, we write $[\mathbf{for} \vec{z} < \vec{e} \rightarrow E]$ for $[\mathbf{for} z_1 < e_1 \rightarrow [\mathbf{for} z_2 < e_2 \rightarrow \dots [\mathbf{for} z_n < e_n \rightarrow E] \dots]]$ and $\text{fv}(\vec{z})$ for $\{z_1, \dots, z_n\}$.

As mentioned above, the translation only uses linear continuations of the form $K = \lambda E.(c \triangleright x : T \ell \text{ viz } E) :: \mathbb{T}$, where \mathbb{T} does not depend on E . If K has the above form, we write $\text{fv}(K)$ for $\text{fv}(T) \cup \text{fv}(\mathbb{T})$.

Translation of regression parameters to Tabular

$$\begin{aligned}
& \llbracket \ell; \vec{e}; \vec{f}; \vec{v}; v\{\alpha \sim r\}; K \rrbracket^\dagger \triangleq \\
& \quad \llbracket \mathbf{static}; \vec{f}; \llbracket; \llbracket; r; \lambda E.(\alpha \triangleright y : \mathbf{real} ! \mathbf{rnd}[\vec{f}] \mathbf{static output } E) :: (K \langle y; \vec{e}; \vec{v}; v\{\alpha \sim r\} \rangle) \rrbracket^\dagger \\
& \quad \text{where } y \notin \text{fv}(K) \cup \text{fv}(\vec{v}) \cup \text{fv}(\vec{e}) \cup \text{fv}(v) \text{ and } \vec{v} = [v_1, \dots, v_n] \\
& \quad \text{and } \langle y; \vec{e}; \vec{v}; v\{\alpha \sim r\} \rangle = [\mathbf{for} \vec{z} < \vec{e} \rightarrow \llbracket v \rrbracket \vec{z} \times y[\llbracket v_n \rrbracket \vec{z}] \dots [\llbracket v_1 \rrbracket \vec{z}]] \\
& \quad \text{and } \text{fv}(\vec{z}) \cap (\{y\} \cup \text{fv}(v) \cup \text{fv}(\vec{v})) = \emptyset \\
& \llbracket \ell; \vec{e}; \vec{f}; \vec{v}; r|v; K \rrbracket^\dagger \triangleq \\
& \quad \llbracket \ell; \vec{e}; f :: \vec{f}; v :: \vec{v}; r; K \rrbracket^\dagger \quad \text{where } \Gamma; \vec{e}; \ell \vdash v : \mathbf{mod}(f) ! \mathbf{det} \\
& \llbracket \ell; \vec{e}; \vec{f}; \vec{v}; D[\hat{e}_1, \dots, \hat{e}_{m'}](u_1, \dots, u_m); K \rrbracket^\dagger \triangleq \\
& \quad K[\mathbf{for} \vec{z} < \vec{e} \rightarrow D[\hat{e}_1, \dots, \hat{e}_{m'}](\llbracket u_1 \rrbracket \vec{z}, \dots, \llbracket u_m \rrbracket \vec{z})] \\
& \llbracket \ell; \vec{e}; \vec{f}; \vec{v}; r_1 + r_2; K \rrbracket^\dagger \triangleq \\
& \quad \llbracket \ell; \vec{e}; \vec{f}; \vec{v}; r_1; \lambda E_1.(_ \triangleright y : \mathbf{real} ! \mathbf{rnd}[\vec{e}] \ell \mathbf{local } E_1) :: \\
& \quad \quad \llbracket \ell; \vec{e}; \vec{f}; \vec{v}; r_2; \lambda E_2.(_ \triangleright z : \mathbf{real} ! \mathbf{rnd}[\vec{e}] \ell \mathbf{local } E_2) :: K [\mathbf{for} \vec{z} < \vec{e} \rightarrow y[\vec{z}] + z[\vec{z}]] \rrbracket^\dagger \\
& \quad \text{where } y \notin \text{fv}(K) \cup \text{fv}(r_2) \cup \text{fv}(\vec{v}) \cup \text{fv}(\vec{e}) \cup \text{fv}(\vec{f}) \text{ and } z \notin \text{fv}(K) \cup \text{fv}(\vec{v}) \cup \text{fv}(\vec{e}) \\
& \quad \text{and } \text{fv}(\vec{z}) \cap \{y, z\} = \emptyset \\
& \llbracket \ell; \vec{e}; \vec{f}; \vec{v}; y \sim r \text{ in } r'; K \rrbracket^\dagger \triangleq \\
& \quad \llbracket \mathbf{static}; \vec{f}; \llbracket; \llbracket; r; \lambda E.(_ \triangleright y : \mathbf{real} ! \mathbf{rnd}[\vec{f}] \mathbf{static local } E) :: \llbracket \ell; \vec{e}; \vec{f}; \vec{v}; r'; K \rrbracket^\dagger \rrbracket^\dagger \\
& \quad \text{where } y \notin \text{fv}(K) \cup \text{fv}(\vec{v}) \cup \text{fv}(\vec{e}) \cup \text{fv}(\vec{f})
\end{aligned}$$

The reason why we need to use continuations is that when processing a given regression r , we cannot immediately compute its output expression, as we do not know the local names of the parameters on which this expression depends, which are chosen dynamically. Using a continuation is a way of delaying the computation of the output expression until the local names of these parameters are known.

In the case of a modelled predictor $v\{\alpha \sim r\}$, we first need to construct a new continuation K' from K . In K' , the expression E , to be computed later, is put in the

first column of the table, defining the parameter α modelled by r . Note that we cannot translate the output expression of regression r yet, because, as stated above, local names of parameters (such as y here) are defined on the fly, and we do not know the names of variables appearing in the translation of the output expression of r before recursing into r . The rest of the table is obtained by passing the output expression of the current regression $v\{\alpha \sim r\}$ down to the continuation K . Having constructed the new continuation K' , we compute recursively the translation of r with this continuation. The previous parameter dimensionality \vec{f} becomes the output dimensionality, as the output expression of r models the parameter α .

In case the regression is of the form $r|v$, the predictor v is added to the list of categorical predictors and its bound f to the list defining parameter dimensionality in the recursive call.

If the regression is a noise term $D[\hat{e}_1, \dots, \hat{e}_m](\llbracket u_1 \rrbracket \vec{z}, \dots, \llbracket u_m \rrbracket \vec{z})$, then its output expression is a nested **for**-loop of dimensionality \vec{e} , whose every component is a random draw from D with the parameters obtained by accessing the elements of cubes defined by u_1, \dots, u_m at the given index \vec{z} .

In the case of a sum $r_1 + r_2$ of two regressions, we recursively call the translation function to translate both expressions r_1 and r_2 and put the translation of r_2 in the continuation of r_1 , to combine the tables resulting from translating these regressions. The outputs of regressions r_1 and r_2 are stored in separate columns with local names y and z and the output expression of $r_1 + r_2$, passed down to K , is then a pointwise sum of the expressions stored in y and z .

Finally, if the regression is a local binding of the form $y \sim r$ **in** r' , it is evaluated by translating r' first (with current continuation K), treating y as a free variable (which cannot be α -converted), and then putting the resulting table in the continuation of r , where it will be preceded by the column with local name y defining the output expression of r . This has the desired effect of binding the return expression of r in r' . Note that in the translation of r , \vec{f} becomes the output dimensionality, because y is supposed to be at the level of parameters, rather than top-level predictors, in r' .

The translation shown above does not produce optimal code, as the rule translating sums of regressions needlessly creates additional columns y and z , storing multidimensional arrays which then need to be accessed to compute the pointwise sum of output expressions of regressions. The translation was defined this way because the invariant that the argument expression E is only used in the first column of the body of a continuation and that output expressions of regressions are computed in one go, instead

of the element at each index being computed separately, simplifies reasoning about the translation, and specifically the proof of type soundness, presented later in this chapter.

In the implementation of a Fabular compiler, it would be easy to optimise the translation by inlining the expressions E_1 and E_2 , which define loops of the form $[\text{for } \vec{z} < \vec{e} \rightarrow \hat{E}_1[\vec{z}]]$ and $[\text{for } \vec{z} < \vec{e} \rightarrow \hat{E}_2[\vec{z}]]$ respectively, and then simplifying $E_1[\vec{z}]$ to $\hat{E}_1[\vec{z}]$, and similarly for E_2 .

5.4.3 Examples

Recall again the two examples from Section 5.2. The last regression for the cheese sales database, modelling sales volume, has three predictors: continuous predictor price and discrete predictors city and chain. These predictors need to be defined in the Fabular schema before the regression.

We first define a Tabular schema to which the given database conforms. We define empty tables **Cities** and **Chains** (discarding the auxiliary string columns with names) and a main table **Sales**, in which we define the city and chain predictors as input columns, linking to the aforementioned tables. We then define price as a real-valued input column. With these columns in place, we can finally define the regression, modelling the observed volume column. Columns are referenced in regressions by their local, not field, names, but in the examples we use same field and local names for input columns.

table Cities			
table Chains			
table Sales			
city ▷ city	link(Cities)!det	input	
chain ▷ chain	link(Chains)!det	input	
price ▷ chain	real!det	input	
volume ▷ volume	real!rnd	output	~(1{a} city)+ (price{b} chain)+ ?

This Fabular table reduces to the following Core Tabular schema (after applying the translation and reducing the dummy function application produced by (RED REGR)):

table Cities			
table Chains			
table Sales			
city ▷ city	link(Cities)!det	input	
chain ▷ chain	link(Chains)!det	input	
price ▷ chain	real!det	input	
a ▷ x	real!rnd[sizeof(Cities)]	static output	[for z < sizeof(Cities)→ Gaussian(0, s_large ^ 2)]
_ ▷ p	real!rnd	local	1*x[city]
b ▷ y	real!rnd[sizeof(Chains)]	static output	[for z < sizeof(Chains)→ Gaussian(0, s_large ^ 2)]
_ ▷ q	real!rnd	local	price*y[chain]
_ ▷ z	real!rnd	static local	GammaSR(1, s_large)
_ ▷ s	real!rnd	local	GaussianMP(0, z)
_ ▷ t	real!rnd	local	q + s
volume ▷ volume	real!rnd	output	p + t

The radon example is similar. We have two tables, **Counties** and **Measurements**, the latter being the top-level table in which the regression is defined. The difference is that in this example, the **Counties** table has its own real-valued input, uranium. The radon model can be encoded in Fabular as follows:

table Counties			
uranium ▷ uranium	real!det	input	
table Measurements			
county ▷ county	link(Counties)!det	input	
floor ▷ floor	real!det	input	
uranium ▷ uranium	real!det[sizeof(Counties)]	static output	[for i < sizeof(Counties)→ i:Counties.uranium]
activity ▷ activity	real!rnd	output	~(1{a ~ (1{ g } + uranium{ h })} county)+ (floor{b})+ ?

Since predictors in regressions can only be locally defined variables, the column uranium has to be copied to the **Measurements** table as a **static** array. While having to do so may seem unnecessary, this is a design choice, made to simplify treatment of regressions conditioned on multiple categorical variables (if we allowed references to other tables, it would not be clear where a variable conditioned on multiple discrete predictors, each with its own table, should be defined).

After inlining the auxiliary variables used in constructing sum expressions, this model takes the following form:

table Counties			
uranium ▷ uranium	real!det	input	
table Measurements			
county ▷ county	link(Counties)!det	input	
floor ▷ floor	real!det	input	
uranium ▷ uranium	real!det[sizeof(Counties)]	static output	[for i < sizeof(Counties) → i:Counties.uranium]
g ▷ z1	real!rnd	static output	Gaussian(0, s_large ^ 2)
h ▷ z2	real!rnd	static output	Gaussian(0, s_large ^ 2)
k ▷ z3	real!rnd	static local	GammaSR(1, s_large)
l ▷ s	real!rnd	static output	[for z < sizeof(Counties) → Gaussian(0, z3)]
a ▷ x	real!rnd[sizeof(Counties)]	static output	[for z < sizeof(Counties) → 1*g + uranium[z]*h + l[z]]
b ▷ y	real!rnd	static output	Gaussian(0, s_large ^ 2)
_ ▷ z	real!rnd	static local	GammaSR(1, s_large)
_ ▷ t	real!rnd	local	GaussianMP(0, z)
activity ▷ activity	real!rnd	output	1*x[county] + floor*y + t

5.4.4 Type Soundness for Fabular

The type soundness result for the reduction of Tabular schema to Core Tabular extends to the reduction of Fabular to Core Tabular. We present an outline of the proof here.

To simplify the proofs, we first define the following admissible typing rules for cube expressions:

Additional typing rules for Tabular expressions

(CUBE ITER) (where $\vec{e} = [e_1, \dots, e_n]$, $\vec{z} = [z_1, \dots, z_n]$)

$\Gamma, z_1 : \mathbf{mod}(e_1) ! \mathbf{det}, \dots, z_n : \mathbf{mod}(e_n) ! \mathbf{det} \vdash^{pc} E : T$

$\Gamma \vdash^{pc} [\mathbf{for} \vec{z} < \vec{e} \rightarrow E] : T[\vec{e}]$

(CUBE INDEX) (where $\vec{e} = [e_1, \dots, e_n]$, $\vec{F} = [F_1, \dots, F_n]$)

$\Gamma \vdash^{pc} E : T[\vec{e}]$

$\Gamma \vdash^{pc} F_i : \mathbf{mod}(e_i) ! spc$

$\Gamma \vdash^{pc} E[\vec{F}] : T \vee spc$

To save space, we write $\Gamma, \vec{z} : \mathbf{mod}(\vec{e})$ for $\Gamma, z_1 : \mathbf{mod}(e_1) ! \mathbf{det}, \dots, z_n : \mathbf{mod}(e_n) ! \mathbf{det}$.

Lemma 15 *The rules (CUBE ITER) and (CUBE INDEX) are admissible.*

Proof: Admissibility of both rules can be proven by induction on the size of \vec{e} . ■

Lemma 16 (Derived judgments) • *If $\Gamma; \vec{e}; \ell \vdash v : T$ and $\vec{e} = [e_1, \dots, e_n]$, then*

$\Gamma \vdash^{\mathbf{static}} e_i : \mathbf{int} ! \mathbf{det}$ for all $i \in 1..n$.

- If $\Gamma; \vec{e}; \vec{f}; \ell \vdash r : \Pi$ and $\vec{e} = [e_1, \dots, e_n]$ and $\vec{f} = [f_1, \dots, f_m]$, then $\Gamma \vdash^{\text{static}} e_i : \text{int! det}$ for all $i \in 1..n$ and $\Gamma \vdash^{\text{static}} f_j : \text{int! det}$ for all $j \in 1..m$.

Proof: By induction on the derivation of $\Gamma; \vec{e}; \ell \vdash v : T$ and $\Gamma; \vec{e}; \vec{f}; \ell \vdash r : \Pi$, respectively. ■

Lemma 17 If $\Gamma; \vec{e}; \ell \vdash v : T$ and $\vec{e} = [e_1, \dots, e_n]$ and $\vec{z} = [z_1, \dots, z_n]$ then $\Gamma, \vec{z} : \ell \text{ mod}(\vec{e}) \vdash^\ell \llbracket v \rrbracket \vec{z} : T$

Proof: By induction on the derivation of $\Gamma; \vec{e}; \ell \vdash v : T$. ■

Lemma 18 If $\Gamma; \vec{e}; \vec{f}; \ell \wedge pc \vdash r ! \Pi$ and $K = \lambda E. (c \triangleright x : \text{real! rnd}[\vec{e}] \ell \text{ viz } E) :: \mathbb{T}$ and $\Gamma, x : \ell \wedge pc \text{ real! rnd}[\vec{e}] \vdash^{pc} \mathbb{T} : Q$ and $\Gamma; \vec{e}; \ell \vdash v_i : \text{mod}(f_i) ! \text{det}$ for all $i \in 1..n$, then

- If $\text{viz} = \text{output}$, then $\Gamma \vdash^{pc} \llbracket \ell; \vec{e}; \vec{f}; \vec{v}; r; K \rrbracket^\dagger : \llbracket \Pi \rrbracket @ ((c \triangleright x : \text{real! rnd}[\vec{e}] (\ell \wedge pc) \text{ output}) :: Q)$
- If $\text{viz} = \text{local}$, then $\Gamma \vdash^{pc} \llbracket \ell; \vec{e}; \vec{f}; \vec{v}; r; K \rrbracket^\dagger : \llbracket \Pi \rrbracket @ Q$

Proof: By induction on the derivation of $\Gamma; \vec{e}; \vec{f}; \ell \wedge pc \vdash r ! \Pi$, with appeal to Lemma 17. Details in Appendix D ■

Proposition 4 (Type preservation for Fabular) (1) If $\Gamma \vdash^{pc} M : Q$ and $M \rightarrow M'$, then $\Gamma \vdash^{pc} M' : Q$

(2) If $\Gamma \vdash^{\text{inst}} \mathbb{T} : Q$ and $\mathbb{T} \rightarrow \mathbb{T}'$, then $\Gamma \vdash^{\text{inst}} \mathbb{T}' : Q$

(3) If $\Gamma \vdash \mathbb{S} : \text{Sty}$ and $\mathbb{S} \rightarrow \mathbb{S}'$, then $\Gamma \vdash \mathbb{S}' : \text{Sty}$.

Proof:

We only need to extend the proof of Proposition 1 with the case (RED REGR) in part 2. In this case, we have $M = \sim r + ?$ and $M' = \llbracket \text{inst}; []; []; []; r; K \rrbracket^\dagger []$, where $K = \lambda E. (_ \triangleright x : \text{real! rnd inst local } E)$

$:: (_ \triangleright y : \text{real! rnd static local } \text{GammaFromShapeAndRate}(0, s_{\text{large}})) ::$

$(\text{ret} \triangleright z : \text{real! rnd inst output } \text{GaussianFromMeanAndPrecision}(x, y)).$

The judgment $\Gamma \vdash^{pc} \sim r + ? : Q$ must have been derived with (MODEL REGRESSION), so $\Gamma; \emptyset; []; []; pc \vdash (r + ?) ! \Pi$ and $Q = \llbracket \Pi \rrbracket @ [(\text{ret} \triangleright y : \text{real! rnd } pc \text{ output})]$.

By inversion of typing, we have $\Gamma; \emptyset; []; []; pc \vdash r ! \Pi$ and $\Gamma; \emptyset; []; []; pc \vdash ? ! \emptyset$, as ? does not define any parameters.

Now, it is easy to check that

$$\begin{aligned} & \Gamma, x :^{pc} \mathbf{real} ! \mathbf{rnd} \vdash^{pc} (_ \triangleright y : \mathbf{real} ! \mathbf{rnd} \mathbf{static} \mathbf{local} \text{GammaFromShapeAndRate}(0, s_{large})) :: \\ & (\text{ret} \triangleright z : \mathbf{real} ! \mathbf{rnd} \mathbf{inst} \mathbf{output} \text{GaussianFromMeanAndPrecision}(x, y)) \\ & : [(\text{ret} \triangleright y : \mathbf{real} ! \mathbf{rnd} \mathbf{pc} \mathbf{output})]. \end{aligned}$$

Hence, by Lemma 18, $\Gamma \vdash^{pc} [\mathbf{inst}; []; []; []; r; K]^\dagger : [\Pi] @ [(\text{ret} \triangleright y : \mathbf{real} ! \mathbf{rnd} \mathbf{pc} \mathbf{output})]$.

Since all columns in $[\Pi]$ are at **static** level (by the definition of $[\Pi]$), we have $\Gamma \vdash^{pc} [] : Q \rightarrow Q$, and so by (MODEL APPL), $\Gamma \vdash^{pc} [\mathbf{inst}; []; []; []; r; \lambda E. (\text{ret} \triangleright y : \mathbf{real} ! \mathbf{rnd} \ell \mathbf{output} E)]^\dagger [] : [\Pi] @ [(\text{ret} \triangleright y : \mathbf{real} ! \mathbf{rnd} \mathbf{pc} \mathbf{output})]$, as required. ■

Proposition 5 (Progress for Fabular) (1) *If $\Gamma \vdash^{pc} \mathbb{T} : Q$ then either $\text{Core}(\mathbb{T})$ or there is \mathbb{T}' such that $\mathbb{T} \rightarrow \mathbb{T}'$.*

(2) *If $\Gamma \vdash^{pc} \mathbb{S} : \text{Sty}$ then either $\text{Core}(\mathbb{S})$ or there is \mathbb{S}' such that $\mathbb{S} \rightarrow \mathbb{S}'$.*

Proof:

If $K = \lambda E. (_ \triangleright x : \mathbf{real} ! \mathbf{rnd} \mathbf{inst} \mathbf{local} E)$

$:: (_ \triangleright y : \mathbf{real} ! \mathbf{rnd} \mathbf{static} \mathbf{local} \text{GammaFromShapeAndRate}(0, s_{large})) ::$

$(\text{ret} \triangleright z : \mathbf{real} ! \mathbf{rnd} \mathbf{inst} \mathbf{output} \text{GaussianFromMeanAndPrecision}(x, y)),$

it is easy to see that $[\mathbb{T}; []; []; []; r; K]^\dagger []$ exists if the regression r is well-typed in Γ . Hence, the progress property for Fabular follows immediately from the same property for Tabular. ■

Proposition 6 (Termination for Fabular) *There does not exist an infinite chain of reductions $\mathbb{S}_1 \rightarrow \mathbb{S}_2 \rightarrow \dots$*

Proof: To adapt the proof of termination to Fabular, it is enough to extend the metric m to regressions, setting $m(r)$ to $k + 1$, where k is the number of columns in the table obtained by reducing r , equal to the number of parameters in r plus the number of local variable definitions plus two times the number of regression additions (each addition generates two local columns) plus one output column (ret). ■

Theorem 3 *If \mathbb{S} is a Fabular schema and $\emptyset \vdash \mathbb{S} : \text{Sty}$, then $\mathbb{S} \rightarrow^* \mathbb{S}'$ for some unique \mathbb{S}' such that $\text{Core}(\mathbb{S}')$ and $\emptyset \vdash \mathbb{S} : \text{Sty}$.*

Proof: The proof is the same as the proof of type soundness of Tabular (Theorem 1), using Propositions 4, 5 and 6. ■

5.5 Conclusions

In this chapter, we have defined a compositional hierarchical linear regression calculus, which extends the languages of formulas used by the packages `lm` and `lmer` in R. We have defined the syntax and type system of the calculus, and embedded the calculus in Tabular, a probabilistic language with a rigorously-defined semantics. We have defined a translation from this extended form of Tabular to Core Tabular and proven that this translation is type-sound.

Limitations of Fabular and Future Work Because of the requirement that the top-level regression must have a Gaussian noise, Fabular does not support generalised linear models with arbitrary link functions. This is arguably the biggest limitation of the language and removing it is an important direction of future work. Another limitation is that Fabular only supports real-valued variables. Adding support for discrete distributions, such as the Poisson distribution, could also be useful.

Other possible directions of future work include automated search for models fitting the data, in the style of [Nori et al., 2015], and automatic plot generation from inference results (which we have already implemented for a restricted set of models for an earlier version of the regression calculus).

Individual Contributions

The paper on which this chapter was based [Borgström et al., 2016] was mostly written in a team effort. However, the updated syntax of the regression calculus presented here (with different treatment of local variables to ensure that variables are α -convertible), updated type system for Fabular and the modified, simplified translation to Core Tabular are my own work. The proof of Theorem 3, showing correctness of the translation of Fabular to Core Tabular, was not included in the paper and is also entirely my own work.

Chapter 6

Semantics of a Lambda Calculus with Continuous Distributions

Acknowledgement This chapter is based on the paper “A Lambda Calculus Foundation for Universal Probabilistic Programming”[Borgström et al., 2016] published at the 2016 International Conference on Functional Programming (ICFP). The paper was joint work with Johannes Borgström, Ugo Dal Lago and Andrew D. Gordon.

In many popular probabilistic languages, including Tabular presented in Chapter 4, programs are interpreted as factor graphs, with a bounded number of random variables. While such languages allow ease of use and efficient inference in many commonly used models, their expressive power is limited. In particular, they do not allow models which do not define distributions on fixed sets of parameters, sometimes called *non-parametric models*. Because of the need to overcome this limitation, a new class of *Universal* probabilistic programming languages, based on functional or procedural Turing-complete languages, has sprung into existence. Imperative universal languages include the R2 language [Nori et al., 2014], while functional ones include Church [Goodman et al., 2008], which pioneered universal probabilistic programming, and its descendants Venture [Mansinghka et al., 2014] and Anglican [Tolpin et al., 2015].

Church is a probabilistic version of Scheme and supports recursion and higher-order functions. A Church program consists of a sequence of (possibly recursive) definitions, followed by the output expression e_q and a condition e_c which must be satisfied for the given run to be valid:

```
(query (define  $x_1$   $e_1$ ) ... (define  $x_n$   $e_n$ )  $e_q$   $e_c$ )
```

For example, the following program defines the geometric distribution (which is the distribution on the number of consecutive flips of a biased coin which come up heads):

```
(query
  (define flip (lambda (p) (< (rnd) p)))
  (define geometric
    (lambda (p)
      (if (flip p) 0 (+ 1 (geometric p))))))
  (define n (geometric .5))
  n
  (> n 2)
)
```

The first **define** statement defines the flip of a coin with bias p (between 0 and 1) by means of the `rnd` function returning a sample from the uniform distribution on the unit interval. The second statement defines a recursive function which samples from flip with the given bias p until the it returns true, and returns the number of times false was drawn. This function implements a draw from the geometric distribution with bias p . Finally, the third statement draws the value of variable n from the geometric distribution with bias 0.5. This value is then returned as the output expression, and restricted to be greater than 2 by the conditioning statement at the end.

Every valid run of this program yields an integer greater than 2, drawn from the geometric distribution with bias 0.5. This means that the distribution on the output values is the geometric distribution conditional on the output value being bigger than 2.

This chapter presents a new approach to defining the semantics of higher-order functional probabilistic languages. It consists of the following parts:

(1) Syntax of an untyped functional calculus:

In order to avoid the complexities of working with a full-featured probabilistic language, we define an untyped probabilistic λ -calculus, capable of encoding the core of Church. The key features of the calculus are draws from primitive distributions and hard and soft conditioning by means of the `fail` exception and a score operator, respectively. We provide a translation of Church programs to the core calculus to demonstrate its expressiveness. To simplify presentation, we restrict our attention to programs with only continuous random draws, but this restriction could easily be lifted.

(2) Semantics of the calculus:

We endow the calculus with a *sampling-based* operational semantics, inspired by [Nori et al., 2013], who define an operational semantics of a procedural while-language, which reduces a program to a state (valuation of variables) and a corresponding weight, given a fixed list of outcomes of random draws. We define a judgment $M \Downarrow_w^s G$ which means that given the linear trace (sequence of random values) s , the expression M reduces to the generalised value G (which can be a value or an exception) with *weight* w . We also provide a small-step semantics, useful in the following chapter, defining the judgment $(M, w, s) \rightarrow (M', w', s')$, which states that the expression M , together with the initial weight w , reduces with trace s in one step to the expression M' , together with the updated weight w' and trace s' .

The main technical result of this chapter, Theorem 4, shows that the small-step semantics is equivalent to the big-step semantics.

We also define a Borel σ -algebra on the terms of the calculus (induced by a straightforward metric on terms) and use the sampling-based semantics to define a *sub-probability distribution* $\llbracket M \rrbracket_{\mathbb{U}}(A)$ on return values of the expression M , by integrating the function induced by the semantics, mapping a trace to a weight, over the set of traces yielding a return value in the given set A . We prove that this distribution is well-defined.

This chapter makes the following contributions:

- (1) Syntax of an untyped λ -calculus with continuous random draws and soft and hard conditioning, capable of encoding Church.
- (2) A sampling-based semantics of the calculus, defining the meaning of a program as a deterministic mapping from a random vector to the output value and weight.
- (3) A function defining the sub-probability distribution on output values of the given program, defined as an integral of the sampling-based semantics.

6.1 A Probabilistic λ -calculus

We start by presenting the probabilistic call-by-value λ -calculus which forms the basis of our work. The calculus is kept small and simple to facilitate reasoning about it,

yet it retains the full expressiveness of functional probabilistic languages, which we demonstrate by providing a translation of Church programs to this calculus. To further simplify the reasoning in the rest of this chapter, we only include continuous random draws in the calculus, as draws from discrete distributions can be encoded by draws from the unit interval and inverse mass functions.

We admit a fixed, countable set of deterministic primitive functions g , each having an arity $|g| > 0$, and a fixed, countable set of random functions D , each with an arity $|D| \geq 0$. We assume that each deterministic primitive g is equipped with an evaluation function $\sigma_g : \mathbb{R}^{|g|} \rightarrow \mathbb{R}$ and each random function D has an underlying probability density function $\text{pdf}_D : \mathbb{R}^{|D|+1} \rightarrow \mathbb{R}$. Furthermore, we assume that all functions σ_g are measurable $\mathcal{R}^{|g|}/\mathcal{R}$ and all densities pdf_D measurable $\mathcal{R}^{|D|+1}/\mathcal{R}$.

Syntax of the core calculus

$V ::=$	value
c	real-valued constant
x	variable
$\lambda x.M$	lambda-abstraction (x bound in M)
$M, N ::=$	expression
V	value
$M N$	application
$D(V_1, \dots, V_{ D })$	(continuous) random draw
$g(V_1, \dots, V_{ g })$	deterministic function
$\text{if } V \text{ then } M \text{ else } N$	conditional
$\text{score}(V)$	soft conditioning
fail	exception

The only constants in the language are lambda-abstractions and real numbers—we assume that `true` and `false` are encoded as 1 and 0, respectively. The lambda-abstraction $\lambda x.M$ binds the variable x in M . As usual, we identify expressions up to alpha-conversion of bound variables.

Primitive functions include the usual arithmetic operators on real numbers ($+$, \times etc.) and comparisons. Random functions include the `rnd` primitive drawing a number from the uniform distribution on the unit interval (with density $\text{pdf}_{\text{rnd}}(c) = 1$ if $c \in [0, 1]$ and 0 otherwise) and the usual Gaussian distribution $\text{Gaussian}(m, v)$, with density $\text{pdf}_{\text{Gaussian}}(m, v, c) = e^{-\frac{(c-m)^2}{2v}} \sqrt{2v\pi}$ if $v > 0$ and 0 otherwise. Only real-valued

distributions are supported, so draws from multivariate distributions, such as multivariate Gaussian, must be simulated. Note that we assume that all deterministic functions and densities of random primitives are total on $\mathcal{R}^{|g|}$ or $\mathcal{R}^{|D|+1}$, and a value must be returned even if the arguments do not make sense, like in the Gaussian example above.

The exception `fail` is used for hard conditioning and, when returned as an output value of an expression, means that a constraint was not satisfied in the given run of the program. The `score` operator is used for soft conditioning. It takes as argument a real value from the unit interval and multiplies the weight of the current trace by its argument, returning a dummy value. Intuitively, `score` assigns higher probability to combinations of random variables which make its argument larger. Because the arguments to `score` are bounded by 1, this form of soft conditioning does not allow conditioning on the outcome of an arbitrary random draw. For instance, it is, in general, not possible to condition on the outcome of a draw from a Gaussian distribution, because the Gaussian density can admit values greater than 1 and the expression `score(pdfGaussian(μ, σ^2, c))` will result in failure if the density of $\text{Gaussian}(\mu, \sigma^2)$ is greater than 1 at c . The reason for this restriction is discussed in Section 6.3.2.

We denote by Λ the set of all terms and by $C\Lambda$ the set of *closed* terms—that is, terms in which all variables are bound by lambda-abstractions. We write \mathcal{V} for the set of all closed values and we define the set of *generalised values* to be the set $\mathcal{GV} = \mathcal{V} \cup \{\text{fail}\}$. We also define a class of *erroneous redexes*, ranged over by variables T, R, \dots , which are not well-formed expressions and cannot be reduced. Specifically, erroneous redexes are expressions of the form:

- $c\ M$
- $D(V_1, \dots, V_{|D|})$ where at least one of the arguments $V_1, \dots, V_{|D|}$ is a λ term
- $g(V_1, \dots, V_{|g|})$ where at least one of the arguments $V_1, \dots, V_{|g|}$ is a λ term
- `if V then M else N` where $V \notin \{\text{true}, \text{false}\}$
- `score(V)` where $V \notin (0, 1]$

To simplify the semantics, we only allow values to be used as primitive function arguments, guards in conditionals and arguments to `score`, but the general forms of these constructs, allowing arbitrary expressions to be used, can be derived in the usual way using the **let** binding, which can itself be encoded as a function application:

Derived constructs

$\text{let } x = M \text{ in } N \triangleq (\lambda x. N) M$
 $D(M_1, \dots, M_{|D|}) \triangleq \text{let } x_1 = M_1 \text{ in } \dots \text{let } x_{|D|} = M_{|D|} \text{ in } D(x_1, \dots, x_{|D|})$
 where $x_1, \dots, x_{|D|}$ distinct and $x_1, \dots, x_{|D|} \notin \text{fv } M_1 \cup \dots \cup \text{fv } M_{|D|}$
 $g(M_1, \dots, M_{|g|}) \triangleq \text{let } x_1 = M_1 \text{ in } \dots \text{let } x_{|g|} = M_{|g|} \text{ in } g(x_1, \dots, x_{|g|})$
 where $x_1, \dots, x_{|g|}$ distinct and $x_1, \dots, x_{|g|} \notin \text{fv } M_1 \cup \dots \cup \text{fv } M_{|g|}$
 $\text{score}(M) \triangleq \text{let } x = M \text{ in } \text{score}(x)$

6.1.1 Big-step Sampling-based Semantics

In this section, we define the big-step sampling-based semantics for the core calculus, defining the judgment $M \Downarrow_w^s G$, which means that the expression M reduces with trace s to a generalised value G with weight w . If $G \in \mathcal{V}$ then the trace s is considered valid and if $G = \text{fail}$, then s must have failed to satisfy a hard constraint and is not considered a valid trace.

Formally, a trace s is defined to be a finite list of real numbers $[s_0, s_1, \dots, s_n]$ of arbitrary length n . Note that in contrast to, for example, [Park et al., 2005], who treat traces as infinite streams, we only consider *finite* traces, and in the derivation of $M \Downarrow_w^s G$, s is *precisely* the list of random values used.

The big-step sampling-based semantics is defined to be the least relation closed under the following rules:

Sampling-based semantics: $M \Downarrow_w^s G$

(EVAL VAL)	(EVAL RANDOM)	(EVAL RANDOM FAIL)	(EVAL PRIM)
$G \in \mathcal{G}\mathcal{V}$	$w = \text{pdf}_D(\vec{c}, c)$	$\text{pdf}_D(\vec{c}, c) = 0$	
$\frac{G \in \mathcal{G}\mathcal{V}}{G \Downarrow_1^\square G}$	$\frac{w > 0}{D(\vec{c}) \Downarrow_w^{[c]} c}$	$\frac{\text{pdf}_D(\vec{c}, c) = 0}{D(\vec{c}) \Downarrow_0^{[c]} \text{fail}}$	$\frac{}{g(\vec{c}) \Downarrow_1^\square \sigma_g(\vec{c})}$
(EVAL APPL)	(EVAL APPL RAISE1)	(EVAL APPL RAISE2)	
$M \Downarrow_{w_1}^{s_1} \lambda x. P \quad N \Downarrow_{w_2}^{s_2} V$	$M \Downarrow_w^s \text{fail}$	$M \Downarrow_w^s c$	
$\frac{P\{V/x\} \Downarrow_{w_3}^{s_3} G}{M N \Downarrow_{w_1 \cdot w_2 \cdot w_3}^{s_1 @ s_2 @ s_3} G}$	$M N \Downarrow_w^s \text{fail}$	$M N \Downarrow_w^s \text{fail}$	

(EVAL APPL RAISE3)			(EVAL IF TRUE)		
$M \Downarrow_{w_1}^{s_1} \lambda x. P$			$M \Downarrow_w^s G$		
$N \Downarrow_{w_2}^{s_2} \text{fail}$					
$M N \Downarrow_{w_1 \cdot w_2}^{s_1 @ s_2} \text{fail}$			$\text{if true then } M \text{ else } N \Downarrow_w^s G$		
(EVAL IF FALSE)			(EVAL SCORE)	(EVAL FAIL)	
$N \Downarrow_w^s G$			$c \in (0, 1]$	$T \text{ is an erroneous redex}$	
$\text{if false then } M \text{ else } N \Downarrow_w^s G$			$\text{score}(c) \Downarrow_c^\square \text{true}$	$T \Downarrow_1^\square \text{fail}$	

In the rules above, \vec{c} is a shorthand for $c_1, \dots, c_{|g|}$ in (EVAL PRIM) and $c_1, \dots, c_{|D|}$ in (EVAL RANDOM) and (EVAL RANDOM FAIL).

The (EVAL VAL) rule just returns an expression which already is a value with weight 1. The (EVAL RANDOM) rule evaluates the random draw $D(\vec{c})$ to c , the only component of the trace s , assumed to be the value drawn from the distribution. The weight returned is the value of the density function of $D(\vec{c})$ at c , required to be positive. The (EVAL RANDOM FAIL) rule returns an exception if at the value c , deemed to be the value drawn from $D(\vec{c})$, the density of $D(\vec{c})$ is in fact zero. (EVAL PRIM) evaluates a deterministic function call, without consuming any randomness or changing the weight.

The (EVAL APPL) rule is the standard application rule for the call-by-value lambda calculus, modulo traces and weights. Derivation of each assumption consumes a (possibly empty) random vector and yields a weight—the random vector consumed by (EVAL APPL) is then the concatenation of vectors consumed by subcomputations, and the weight is the product of weights yielded by subcomputations.

The three subsequent rules for applications are necessary to account for the fact that deriving one of the three assumptions of (EVAL APPL) may raise an exception, making it impossible to compute the outcome of an application. The rules (EVAL APPL RAISE1), (EVAL APPL RAISE2) and (EVAL APPL RAISE3) raise an exception when computing the function fails, the function expression is in fact a real constant and when computing the value of the argument fails, respectively.

The rules (EVAL IF TRUE) and (EVAL IF FALSE) are standard. The following rule, (EVAL SCORE), reduces $\text{score}(c)$ to the dummy value `true` with weight c , provided c is a positive real bounded by one—the reason for enforcing this bound is explained later in this chapter. The `score` primitive is expected to be used, for example, in conditionals and `let`-expressions only for its side effect of changing the weight of the current trace, and the return value is discarded. Finally, (EVAL FAIL) reduces an erroneous redex to

fail.

For example, suppose we have the following program M :

$$(\lambda x. \text{Gaussian}(x, 1)) \text{Uniform}()$$

This program samples a value from a Gaussian distribution with mean sampled randomly from the unit interval and variance set to 1. Suppose that we are given the random trace $s = [0.3, 0.7]$. Since $\text{pdf}_{\text{Uniform}}(0.3) = 1$, by (EVAL RANDOM) we have $\text{Uniform}() \Downarrow_1^{[0.3]} 0.3$. Similarly, by (EVAL RANDOM), we have $\text{Gaussian}(0.3, 1) \Downarrow_w^{[0.7]} 0.7$, where $w = \text{pdf}_{\text{Gaussian}}(0.3, 1, 0.7)$. Thus, (EVAL APPL) gives

$$(\lambda x. \text{Gaussian}(x, 1)) \text{Uniform}() \Downarrow_w^{[0.3, 0.7]} 0.7$$

where, again, $w = \text{pdf}_{\text{Gaussian}}(0.3, 1, 0.7)$.

6.1.2 Encoding Church in the Core Calculus

In this section, we define the translation of Church programs to the core calculus.

The syntax of the original presentation of Church, as defined in [Goodman et al., 2008], consists of the following grammar of expressions, definitions and (top-level) stochastic queries:

Syntax of Church

$e ::=$	Expression
c	constant
x	variable
$(g\ e_1 \dots e_n)$	deterministic primitive function
$(D\ e_1 \dots e_n)$	random draw
$(\text{if}\ e_1\ e_2\ e_3)$	conditional
$(\text{lambda}\ (x_1 \dots x_n)\ e)$	lambda abstraction
$(e_1\ e_2 \dots e_n)$	application
$d ::= (\text{define}\ x\ e)$	Definition (possibly recursive)
$q ::= (\text{query}\ d_1 \dots d_n\ e\ e_{\text{cond}})$	Query

The syntax of Church expressions, based on Scheme, is self-explanatory—note that the language, unlike our core calculus, supports functions with multiple arguments and

allows arbitrary expressions as primitive function arguments and guards. A Church program is a *query*, consisting of a sequence of possibly recursive definitions (introduced by the **define** keyword), an output expression and a boolean-valued condition, which must evaluate to true for the given program run to be valid.

In order to translate recursive function definitions to the probabilistic λ -calculus, we need to use the following call-by-value fixpoint operator $fix\ x.M$, defined as

$$fix\ x.M \triangleq \lambda y. N_{fix}\ N_{fix}\ (\lambda x.M)y$$

where

$$N_{fix} = \lambda z. \lambda w. w(\lambda y. ((zz)w)y)$$

The translation of Church to the calculus is defined by the following rules:

Translation of Church

$$\begin{aligned}
\langle c \rangle &= c \\
\langle x \rangle &= x \\
\langle g\ e_1, \dots, e_n \rangle &= \\
&\quad \text{let } x_1 = e_1 \text{ in } \dots \text{let } x_n = e_n \text{ in } g(x_1, \dots, x_n) \\
&\quad \text{where } x_1, \dots, x_n \text{ distinct and } x_1, \dots, x_n \notin \text{fv}(e_1) \cup \dots \cup \text{fv}(e_n) \\
\langle D\ e_1, \dots, e_n \rangle &= \\
&\quad \text{let } x_1 = e_1 \text{ in } \dots \text{let } x_n = e_n \text{ in } D(x_1, \dots, x_n) \\
&\quad \text{where } x_1, \dots, x_n \text{ distinct and } x_1, \dots, x_n \notin \text{fv}(e_1) \cup \dots \cup \text{fv}(e_n) \\
\langle \text{lambda } ()\ e \rangle &= \lambda x. \langle e \rangle \quad \text{where } x \notin \text{fv}(e) \\
\langle \text{lambda } x\ e \rangle &= \lambda x. \langle e \rangle \\
\langle \text{lambda } (x_1 \dots x_n)\ e \rangle &= \lambda x_1. \langle \text{lambda } (x_2 \dots x_n)\ e \rangle \\
\langle e_1\ e_2 \rangle &= \langle e_1 \rangle \langle e_2 \rangle \\
\langle e_1\ e_2 \dots e_n \rangle &= \langle (e_1\ e_2) \dots e_n \rangle \\
\langle \text{if } e_1\ e_2\ e_3 \rangle &= \text{let } x = e_1 \text{ in } (\text{if } x \text{ then } \langle e_2 \rangle \text{ else } \langle e_3 \rangle) \\
&\quad \text{where } x \notin \text{fv}(e_2) \cup \text{fv}(e_3) \\
\langle \text{query } (\text{define } x_1\ e_1) \dots (\text{define } x_n\ e_n)\ e_{out}\ e_{cond} \rangle &= \\
&\quad \text{let } x_1 = (fix\ x_1. \langle e_1 \rangle) \text{ in} \\
&\quad \dots \\
&\quad \text{let } x_n = (fix\ x_n. \langle e_n \rangle) \text{ in} \\
&\quad \text{let } b = e_{cond} \text{ in}
\end{aligned}$$

```
if  $b$  then  $e_{out}$  else fail
```

For completeness, it should be noted that the full Church language also supports *stochastic memoisation* [Goodman et al., 2008]. Unlike memoisation in deterministic languages, which is purely an optimization measure, stochastic memoisation is a semantically significant construct. It amounts to restricting a given random function to always return the same value for the same arguments in a single run of the program—when a function is first called with given arguments, the return value is stored, and when the function is called again with the same arguments, the stored value is returned, without re-evaluating the function.

In Church, memoisation is provided by a special function `mem`, which takes a (possibly random) lambda abstraction and returns its memoised version. It is useful in defining some nonparametric models, such as the Dirichlet Process [Ferguson, 1973], since memoised random functions on integers can be treated as infinite lazy lists of random values.

We could have added support for memoisation in our translation of Church by changing the translation to state-passing style, but decided against doing so, because memoisation is not the main focus of this work and we preferred to keep the encoding simple.

6.1.3 Example: Geometric Distribution

To further explain the sampling-based semantics, let us revisit the geometric distribution example from the introduction. The translation of this example to the core calculus (simplified slightly for readability) takes the following form (recall that `let` is actually syntactic sugar for application).

```
let geometric =  
  (fix g.  
     $\lambda p.$  (let  $z = \text{rnd}()$  in  
      let  $y = (z < p)$  in  
      if  $y$  then 0 else  $1 + (g\ p)$ )) in  
let  $n = \text{geometric } 0.5$  in  
let  $b = n > 1$  in  
if  $b$  then  $n$  else fail
```

Suppose we want to evaluate this program with trace $s = [0.6, 0.7, 0.2]$. That is, if we call this program M , we want to find G , w such that $M \Downarrow_w^{[0.6, 0.7, 0.2]} G$.

After desugaring the let-bindings, by (EVAL APPL) we can substitute the definition of *geometric* in the remainder of the program, without consuming any randomness or changing weight. Then we need to evaluate the call to *geometric*. In the definition of *geometric*, we can unfold the recursion by using the easy to show fact that for any $\lambda x.M$, $M \{ \text{fix } x.M / x \} V \Downarrow_w^s G$ if and only if $(\text{fix } x.M) V \Downarrow_w^s G$. Unfolding the recursion in *geometric* 0.5 yields the following expression:

```

let z = rnd() in
let y = z < 0.5 in
if y then 0 else
  1 + ((let z = rnd() in
        let y = z < 0.5 in
        if y then 0 else 1 + (...))

```

Because of the call-by-value evaluation strategy, calls to `rnd()` are evaluated in sequence, as they appear in the program. Applied to the first random draw, (EVAL RANDOM) gives `rnd()` $\Downarrow_1^{[0.6]} 0.6$ (as the density of `rnd` is constant and equal to 1 on the unit interval). By (EVAL APPL), we can (deterministically) reduce the outermost let-expression, replacing z with 0.6. Then, by (EVAL PRIM), $0.6 < 0.5 \Downarrow_1^\square \text{false}$, so applying (EVAL APPL) again, we can replace y with `false`. This makes the guard of the outermost if-expression false, so by (EVAL IF FALSE), we evaluate the expression by evaluating the else-branch, which takes the same form as the original unfolded body of the call to *geometric*.

We evaluate the next unfolding of the recursion in the same way—this time, we get `rnd()` $\Downarrow_1^{[0.7]} 0.7$, so the guard again evaluates to `false`. However, in the following unfolding, we have `rnd()` $\Downarrow_1^{[0.2]} 0.2$, so this time, by (EVAL PRIM), the guard evaluates to `true` and the recursion ends, with the outcome computed by (EVAL PRIM). In the end, we get *geometric* 0.5 $\Downarrow_1^{[0.6, 0.7, 0.2]} 2$. Since the condition $n > 1$ is clearly satisfied, we can easily derive $M \Downarrow_1^{[0.6, 0.7, 0.2]} 2$ for the full program M .

Note that as the only distribution sampled from is `rnd()`, which has a constant pdf, this program evaluates with weight 1 for every valid trace, regardless of its length and of the return value. This may seem counter-intuitive, because lower values are clearly more likely (specifically, the program evaluates to any $n > 1$ with unnormalised probability $\frac{1}{2^{n+1}}$). However, as described later in this chapter, the probability of a given outcome is computed as an integral of the weight over the set of traces yielding this

outcome, and with respect to the stock measure on traces, sets of traces leading to higher values are “smaller”.

6.1.4 score and Soft Conditioning

In addition to hard conditioning, performed by using the exception `fail`, which rejects all program traces resulting in some Boolean condition not being satisfied, our calculus also supports *soft conditioning*, which allows modifying the weight of a given trace depending on how likely we consider this trace to be. Soft conditioning is frequently used in machine learning to model observations of noisy data—for example, in the Old Faithful eruption model in Section 4.3, we use it to account for the fact that eruptions whose times are closer to the mean of a given cluster are more likely to be in this cluster.

The most general form of soft conditioning allows multiplying the weight of the given trace by an arbitrary positive number, which can be the density of some distribution at a given point. This allows conditioning on the outcomes of arbitrary random draws. For instance, we can use a construct like `score(pdfGaussian(μ, σ^2, c))` to account for the fact that the observed value of a draw from a Gaussian with mean μ and variance σ^2 was c . However, unlike in the semantics of Tabular, where scores could be values of arbitrary density functions, we decided to only allow scores bounded by 1 in the probabilistic λ -calculus, because allowing scores greater than 1 in the presence of recursion could cause well-behaved programs, terminating with probability 1, to have infinite expected weight (as explained in Section 6.3.2).

A very popular kind of soft conditioning, often used in Bayesian linear regression, is assuming that a given observed data point c is a *noisy copy* of some quantity x and was drawn from a Gaussian centred at x . Because of the aforementioned restriction, we cannot implement this directly, but we can still force x to be close to c by multiplying the trace weight by the term $\exp(-(x - c)^2)$, which is always between 0 and 1. This is equivalent to assuming that c was drawn from a Gaussian with mean x and variance $\frac{1}{2}$, up to normalisation.

In principle, this form of soft conditioning can be simulated by hard conditioning, by using rejection sampling, making a given trace more likely to be accepted if a given variable is closer to its expected value. This could be done, for instance, by defining the following operator, using the *flip* function, as defined in the opening example, and the generalised version of *if*:

$$\text{condition } x \text{ c } M \triangleq \text{if flip}(\exp(-(x-c)^2)) \text{ then } M \text{ else fail}$$

Here x is a random variable whose value we want to condition and c is the expected value of x . The *condition* operator draws a sample from the uniform distribution on the unit interval and only allows the execution to proceed if the sample is greater or equal to $\exp(-(x-c)^2)$ which has the effect of rejecting the given program run with probability $1 - \exp(-(x-c)^2)$. Thus, in the context of sampling, this operator has the desired effect of assigning lower probability to traces in which x is further from c . However, this way of performing soft conditioning is very inefficient, since less likely traces are rejected rather than just assigned lower weight—this means that a sampling-based algorithm must generate possibly many more samples to yield meaningful results.

For this reason, our language includes the *score* operator, which only modifies the weight of a given trace, without rejecting any traces. The *condition* function could be redefined using *score* as follows:

$$\text{condition } x \text{ c } M \triangleq \text{let } _ = \text{score}(\exp(-(x-c)^2)) \text{ in } M$$

This version is semantically equivalent to the previous one (which can be shown by integrating the weight, as explained later), but leads to much more efficient inference algorithms. Note that the dummy value returned by *score* is discarded, so we can use a wildcard in *let*.

Example: Linear Regression To illustrate the use of soft conditioning, let us consider the standard Bayesian linear regression model. The model tries to fit a line $y = m \times x + b + \delta$ to the observed data, assuming that the coefficients m , b have Gaussian priors and δ is the noise term. If we consider the noise to be modelled by a Gaussian with mean 0 and variance $\frac{1}{2}$, we can perform the conditioning by multiplying each trace weight by $\exp(-(y - \hat{y})^2)$ for each random value y expected to be \hat{y} , as described above. This can be simulated by rejection sampling, accepting each value y with probability $\exp(-(y - \hat{y})^2)$.

If we assume that we have observed four data points, $(0,0)$, $(1,1)$, $(2,4)$ and $(3,6)$ and that the Gaussian priors of both coefficients m and b have mean 0 and variance 2, we can represent the model as the following program (written in Church syntax), which samples the value of the regression at $x = 4$:

(query


```

(define sqr (lambda (x) (* x x)))
(define squash (lambda (x y) exp (- (sqr (- x y)))))
(define flip (lambda (p) (< (rnd) p)))
(define softeq (lambda (x y) (flip (squash x y))))

(define m (gaussian 0 2))
(define b (gaussian 0 2))
(define f (lambda (x y) (+ (* m x) y)))

(f 4)

(when (softeq (f 0) 0) (softeq (f 1) 1) (softeq (f 2) 4) (softeq (f 3) 6))
)

```

However, applying a sampling-based inference algorithm to this model would result in poor performance, because the model draws four auxiliary random variables to perform the conditioning, and runs for which at least one condition is not satisfied are discarded. In order to overcome this problem, we can transform this model into a semantically equivalent one, by redefining `softeq` to use `score` and modify the weight instead of rejecting a trace:

```

(define softeq (score (squash x y)))

```

In this updated version of the model, only two random values (for m and b) are sampled and no traces are rejected, which leads to more efficient inference.

6.2 Small-step Semantics

Having defined and explained the big-step sampling-based semantics of the core calculus, we now introduce an equivalent small-step semantics. In addition to offering a different view on term evaluation, small-step semantics is more convenient to use in certain proofs in this chapter and is also used in the definition of partial term evaluation in Chapter 7.

We begin by defining the grammars of *evaluation contexts* and *redexes*.

Evaluation contexts E and redexes R

$E ::=$

$[\cdot]$

$$E M$$

$$(\lambda x.M) E$$

$$R ::=$$

$$(\lambda x.M) V$$

$$D(\vec{c})$$

$$g(\vec{c})$$

$$\text{score}(c)$$

$$\text{fail}$$

$$\text{if true then } M \text{ else } N$$

$$\text{if false then } M \text{ else } N$$

$$T$$

Recall that the metavariable T ranges over erroneous redexes. As usual, $E[M]$ denotes the term obtained by plugging M into the unique hole in E .

We call an evaluation context E *closed* if any variable x only occurs in it as a subterm of the term $\lambda x.M$. We let \mathcal{C} be the set of closed contexts. A term M is *reducible* if $M = E[R]$ for some E, R .

Lemma 19 *Every closed term M is either a generalised value or can be split into unique E, R such that $M = E[R]$. Moreover, if $M \notin \mathcal{GV}$ and $R = \text{fail}$, then E is proper (i.e. $E \neq [\cdot]$).*

Proof: By induction on the structure of M . ■

We define *context composition* $E \circ E'$ inductively as follows:

Context composition: $E \circ E'$

$$[\cdot] \circ E' \triangleq E'$$

$$(E M) \circ E' \triangleq (E \circ E') M$$

$$((\lambda x.M) E) \circ E' \triangleq (\lambda x.M)(E \circ E')$$

Lemma 20 $(E \circ E')[M] = E[E'[M]]$.

Proof: By induction on the structure of E . ■

The *deterministic* reduction relation $M \xrightarrow{\text{det}} N$, reducing closed terms other than random draws and score in context, is defined as follows:

Deterministic reduction: $M \xrightarrow{\text{det}} N$

$$\begin{aligned}
& E[g(\vec{c})] \xrightarrow{\text{det}} E[\sigma_g(\vec{c})] \\
& E[(\lambda x.M) V] \xrightarrow{\text{det}} E[M\{V/x\}] \\
& E[\text{if true then } M_1 \text{ else } M_2] \xrightarrow{\text{det}} E[M_1] \\
& E[\text{if false then } M_1 \text{ else } M_2] \xrightarrow{\text{det}} E[M_2] \\
& E[T] \xrightarrow{\text{det}} E[\text{fail}] \\
& E[\text{fail}] \xrightarrow{\text{det}} \text{fail} \quad \text{if } E \neq []
\end{aligned}$$

These deterministic reduction rules are standard for a call-by-value λ -calculus.

Lemma 21 *For every closed M , if $M \xrightarrow{\text{det}} M'$ and $M \xrightarrow{\text{det}} M''$, then $M' = M''$.*

Proof: The assumption $M \xrightarrow{\text{det}} M'$ implies that M is not a generalised value. Hence, Lemma 19 implies that $M = E[R]$ for some unique E, R . If $R = \text{fail}$, then by Lemma 19 E is proper and $E[R]$ can only reduce to fail . If $R \neq \text{fail}$, then by inspection of the reduction rules, if $E[R] \xrightarrow{\text{det}} M'$ and $E[R] \xrightarrow{\text{det}} M''$, then $M' = M'' = E[N]$ for some N uniquely determined by R . ■

Lemma 22 *If $E[R] \xrightarrow{\text{det}} E[N]$, then $R \xrightarrow{\text{det}} N$.*

Proof: By case analysis on the deterministic reduction rules. ■

Lemma 23 *If $R \xrightarrow{\text{det}} N$ and $R \neq \text{fail}$, then for any closed E , $E[R] \xrightarrow{\text{det}} E[N]$.*

Proof: Follows immediately by case analysis on the reduction rules. ■

Lemma 24 *For any closed E and M such that $M \neq E'[\text{fail}]$, if $M \xrightarrow{\text{det}} M'$ then $E[M] \xrightarrow{\text{det}} E[M']$.*

Proof: Since $M \xrightarrow{\text{det}} M'$ implies that M is not a generalised value, by Lemma 19 we have $M = E'[R]$ for some E', R .

By assumption, we have $R \neq \text{fail}$, so by inspection of the reduction rules, we must have $E'[R] \xrightarrow{\text{det}} E'[N]$ for some N .

By Lemma 21, $E'[N] = M'$, and by Lemma 20, $E[M] = (E \circ E')[R]$ and $E[M'] = (E \circ E')[N]$.

Since Lemma 22 implies $R \xrightarrow{\text{det}} N$, by Lemma 23, $(E \circ E')[R] \xrightarrow{\text{det}} (E \circ E')[N]$. This implies $E[M] \xrightarrow{\text{det}} E[M']$, as required. ■

The full small-step sampling-based semantics is defined by a reduction relation on *configurations* (M, w, s) , where M is the current expression, w is the currently accumulated weight and s is the *remaining* (yet to be consumed) random trace. The judgment $(M, w, s) \rightarrow (M', w', s')$ means that the expression M with initial weight w reduces with the random trace s to M' in one step, updating the weight to w' and leaving the suffix s' of trace s unused. We write $c::s$ for a trace whose first element is c and rest is s .

Small-step semantics: $(M, w, s) \rightarrow (M', w', s')$

<p>(RED PURE)</p> $\frac{M \xrightarrow{\text{det}} N}{(M, w, s) \rightarrow (N, w, s)}$	<p>(RED SCORE)</p> $\frac{c \in (0, 1]}{(E[\text{score}(c)], w, s) \rightarrow (E[\text{true}], cw, s)}$
<p>(RED RANDOM)</p> $\frac{w' = \text{pdf}_D(\vec{c}, c) \quad w' > 0}{(E[D(\vec{c})], w, c::s) \rightarrow (E[c], ww', s)}$	<p>(RED RANDOM FAIL)</p> $\frac{\text{pdf}_D(\vec{c}, c) = 0}{(E[D(\vec{c})], w, c::s) \rightarrow (E[\text{fail}], 0, s)}$

The rule (RED PURE) allows reducing an expression deterministically inside a configuration, without affecting the current weight and trace. The (RED SCORE) rule reduces $\text{score}(c)$ (with a valid argument c) inside a context to a dummy value, multiplying the current weight by the argument c . The rule (RED RANDOM) reduces a random draw (inside a context) to the first element c of the trace, removing this element from the trace and multiplying the weight by the density of the given distribution at c , assumed to be positive. Finally, (RED RANDOM FAIL) reduces a random draw to an exception if the corresponding value in the trace is outside the support of the given distribution.

Small-step reduction of configurations is deterministic:

Lemma 25 *If $(M, w, s) \rightarrow (M', w', s')$ and $(M, w, s) \rightarrow (M'', w'', s'')$, then $M' = M''$, $w' = w''$ and $s' = s''$.*

Proof: By case analysis. Since there is no rule that reduces generalised values, $(M, w, s) \rightarrow (M', w', s')$ implies that $M \notin \mathcal{GV}$, so by Lemma 19, $M = E[R]$ for some unique E, R .

- If $(M, w, s) \rightarrow (M', w', s')$ was derived with (RED PURE), then $M = E[R]$, where $R \neq D(\vec{c})$ and $R \neq \text{score}(c)$, which implies that $(M, w, s) \rightarrow (M'', w'', s'')$ must also have been derived with (RED PURE). Hence, we have $w'' = w' = w$, $s'' = s' = s$, $M \xrightarrow{\text{det}} M'$ and $M \xrightarrow{\text{det}} M''$. By Lemma 21, $M'' = M'$, as required.
- If $(M, w, s) \rightarrow (M', w', s')$ was derived with (RED RANDOM), then $M = E[D(\vec{c})]$, $s = c :: s^*$ and $\text{pdf}_D(\vec{c}, c) > 0$. Hence, $(M, w, s) \rightarrow (M'', w'', s'')$ must also have been derived with (RED RANDOM), and so $M'' = M' = E[c]$, $s'' = s' = s^*$ and $w'' = w' = w \text{pdf}_D(\vec{c}, c)$, as required. The (RED RANDOM FAIL) case is analogous.
- If $(M, w, s) \rightarrow (M', w', s')$ was derived with (RED SCORE), then $M = E[\text{score}(c)]$ and $c \in (0, 1]$, so $(M, w, s) \rightarrow (M'', w'', s'')$ must also have been derived with (RED SCORE). Hence $M'' = M' = E[\text{true}]$, $w'' = w' = c \cdot w$ and $s'' = s' = s$.

■

Lemmas 22 and 23 about adding and removing contexts generalise to the pseudo-deterministic reduction of configurations.

Lemma 26 *If $M \neq E'[\text{fail}]$, then for any closed E , if $(M, w, s) \rightarrow (M', w', s')$, then $(E[M], w, s) \rightarrow (E[M'], w', s')$.*

Proof: By case analysis on the derivation of $(M, w, s) \rightarrow (M', w', s')$

- If $(M, w, s) \rightarrow (M', w', s')$ was derived with (RED PURE), then $M \xrightarrow{\text{det}} M'$, so by Lemma 24, $E[M] \xrightarrow{\text{det}} E[M']$, and by (RED PURE), $(E[M], w, s) \rightarrow (E[M'], w', s')$.
- If $(M, w, s) \rightarrow (M', w', s')$ was derived with (RED RANDOM), then $M = E'[D(\vec{c})]$, $M' = E'[c]$, $s = c :: s'$ and $w' = w \text{pdf}_D(\vec{c}, c)$, where $\text{pdf}_D(\vec{c}, c) > 0$. By (RED RANDOM) and Lemma 20, we can derive $(E[M], w, s) \rightarrow (E[M'], w', s')$. Cases (RED RANDOM FAIL) and (RED SCORE) are analogous.

■

Lemma 27 *If $(E[R], w, s) \rightarrow (E[N], w', s')$, then $(R, w, s) \rightarrow (N, w', s')$.*

Proof: By case analysis

- If $(E[R], w, s) \rightarrow (E[N], w', s')$ was derived with (RED PURE), then $E[R] \xrightarrow{\text{det}} E[N]$, so by Lemma 24, $R \xrightarrow{\text{det}} N$, which implies $(M, w, s) \rightarrow (M', w', s')$.
- If $(E[R], w, s) \rightarrow (E[N], w', s')$ was derived with (RED RANDOM), then $R = D(\vec{c})$, $N = c$, $s = c :: s'$ and $w' = w \text{pdf}_D(\vec{c}, c)$, where $\text{pdf}_D(\vec{c}, c) > 0$.

Hence, with (RED RANDOM), we can derive $(D(\vec{c}), w, s) \rightarrow (c, w', s')$

Cases (RED RANDOM FAIL) and (RED SCORE) are analogous. ■

We state several more lemmas about small step reduction, useful in the proof of equivalence of big-step and small-step semantics. First, we note that only (RED RANDOM FAIL) can set the weight to 0, which means that the weight is always positive after successful reductions.

Lemma 28 *If $(M, w, s) \rightarrow (M', w', s')$ was not derived with (RED RANDOM FAIL) and $w > 0$, then $w' > 0$.*

Proof: By inspection. ■

Since the rules do not restrict the initial weight w , multiplying the weight on both sides by any positive number preserves the reduction relation.

Lemma 29 *If $(M, w, s) \rightarrow (M', w', s')$, then for any $w^* \geq 0$, $(M, ww^*, s) \rightarrow (M', w'w^*, s')$*

Proof: By case analysis. ■

Adding the same suffix to the initial and reduced traces preserves the relation.

Lemma 30 *If $(M, w, s) \rightarrow (M', w', s')$, then for any s^* , $(M, w, s@s^*) \rightarrow (M', w', s'@s^*)$*

Proof: By case analysis on the derivation of $(M, w, s) \rightarrow (M', w', s')$. The only interesting cases are (RED RANDOM) and (RED RANDOM FAIL), which modify the trace.

- Case (RED RANDOM): We have $M = E[D(\vec{c})]$ and $M' = E[c]$ for some E , D , \vec{c} and c such that $s = c :: s'$. Moreover, $w' = w\hat{w}$, where $\hat{w} = \text{pdf}_D(\vec{c}, c) > 0$. Hence, for any s^* , $(E[D(\vec{c})], w, c :: (s'@s^*)) \rightarrow (M', w\hat{w}, s'@s^*)$ follows by (RED RANDOM)

- Case (RED RANDOM FAIL): similar.

■

Likewise, removing the unused part of the random trace preserves the reduction relation.

Lemma 31 *If $(M, w, s) \rightarrow (M', w', s')$, then there is s^* such that $s = s^* @ s'$ and $(M, w, s^*) \rightarrow (M', w', [])$*

Proof: By case analysis. The only interesting cases are (RED RANDOM) and (RED RANDOM FAIL).

- Case (RED RANDOM): Like in Lemma 30, we have $M = E[D(\vec{c})]$ and $M' = E[c]$ for some E, D, \vec{c} and c such that $s = c :: s'$ and $w' = w\hat{w}$, where $\hat{w} = \text{pdf}_D(\vec{c}, c) > 0$. Take $s^* = [c]$. By (RED RANDOM), we have $(E[D(\vec{c})], w, [c]) \rightarrow (M', w\hat{w}, [])$.
- Case (RED RANDOM FAIL): similar.

■

We define the *closure* $(M, w, s) \Rightarrow (M', w', s')$ of the small-step semantics inductively by stating that $(M, w, s) \Rightarrow (M', w', s')$ if and only if $(M, w, s) = (M', w', s')$ or $(M, w, s) \rightarrow (M'', w'', s'') \Rightarrow (M', w', s')$ for some M'', w'', s'' . We write $(M, w, s) \rightarrow^k (M', w', s')$ if the configuration (M, w, s) reduces to (M', w', s') in precisely k steps. The multi-step reduction of an expression M to a generalised value G is deterministic for a given trace s .

Lemma 32 *If $(M, w, s) \Rightarrow (G', w', s')$ and $(M, w, s) \Rightarrow (G'', w'', s'')$, then $G' = G''$, $w' = w''$ and $s' = s''$.*

Proof: By induction on the derivation of $(M, w, s) \Rightarrow (G', w', s')$.

- Base case: $(M, w, s) = (G', w', s')$. Generalised values do not reduce, so $G'' = G' = G$, $w'' = w' = w$ and $s'' = s' = s$.
- Induction step: $(M, w, s) \rightarrow (\hat{M}, \hat{w}, \hat{s}) \Rightarrow (G', w', s')$. Since $M \neq G''$, we also have $(M, w, s) \rightarrow (M^*, w^*, s^*) \Rightarrow (G'', w'', s'')$. By Lemma 25, $(M^*, w^*, s^*) = (\hat{M}, \hat{w}, \hat{s})$, and so by induction hypothesis, $(G'', w'', s'') = (G', w', s')$, as required.

■

Expressions on both sides of a multi-step reduction can be placed in an arbitrary closed context, as long as the final expression is not `fail` (because the context is removed when reducing an exception).

Lemma 33 *For any closed E , if $(M, w, s) \Rightarrow (M', w', s')$ and $M' \neq \text{fail}$, then we have $(E[M], w, s) \Rightarrow (E[M'], w', s')$.*

Proof: By induction on the number of steps in the derivation of $(M, w, s) \Rightarrow (M', w', s')$, with appeal to Lemma 26. Since $M' \neq \text{fail}$, no expression in the derivation chain (other than the last one) can be of the form $E'[\text{fail}]$. ■

If the last expression is `fail`, then the original expression will still reduce to `fail` when put in any closed context.

Lemma 34 *For any E , if $(M, w, s) \Rightarrow (\text{fail}, w', s')$ then $(E[M], w, s) \Rightarrow (\text{fail}, w', s')$.*

Proof: By induction on the number of steps in the derivation, using Lemmas 26 and 33. If $E = []$, the result holds trivially, so let us assume $E \neq []$. If $(M, w, s) \Rightarrow (\text{fail}, w', s')$ was derived in 0 steps, then $M = \text{fail}$, $w' = w$ and $s' = s$, so by (RED PURE), $(E[\text{fail}], w, s) \rightarrow (\text{fail}, w, s)$, as required.

If $(M, w, s) \Rightarrow (\text{fail}, w', s')$ was derived in 1 or more steps, then:

- If $M = E'[\text{fail}]$ and $E' \neq []$, then $((E \circ E')[\text{fail}], w, s) \rightarrow (\text{fail}, w', s')$ by (RED PURE).
- Otherwise, there exist \hat{M} , \hat{w} , \hat{s} such that $(M, w, s) \rightarrow (\hat{M}, \hat{w}, \hat{s}) \Rightarrow (\text{fail}, w', s')$, where $M \notin \mathcal{GV}$. By induction hypothesis, $(E[\hat{M}], \hat{w}, \hat{s}) \Rightarrow (\text{fail}, w', s')$ for any E , and by Lemma 26, $(E[M], w, s) \rightarrow (E[\hat{M}], \hat{w}, \hat{s})$. ■

A valid reduction must keep the weight non-negative.

Lemma 35 *If $(M, w, s) \Rightarrow (M', w', s')$ and $w \geq 0$, then $w' \geq 0$.*

Proof: By induction on the number of steps in the derivation.

- If $(M, w, s) \Rightarrow (M', w', s')$ was derived in 0 steps, then $w' = w$, so $w' \geq 0$.

- If $(M, w, s) \Rightarrow (M', w', s')$ was derived in 1 or more steps, then $(M, w, s) \rightarrow (M^*, w^*, s^*) \Rightarrow (M', w', s')$.
 If $(M, w, s) \rightarrow (M^*, w^*, s^*)$ was derived with (RED PURE), then $w^* = w \geq 0$.
 If $(M, w, s) \rightarrow (M^*, w^*, s^*)$ was derived with (RED RANDOM), then $w^* = w \cdot w''$ for some $w'' > 0$, so $w^* \geq 0$.
 If $(M, w, s) \rightarrow (M^*, w^*, s^*)$ was derived with (RED SCORE), then $w^* = w \cdot c$ for some $c > 0$, so $w^* \geq 0$.
 If $(M, w, s) \rightarrow (M^*, w^*, s^*)$ was derived with (RED RANDOM FAIL), then $w^* = 0$.
 In either case, $w^* \geq 0$, so by induction hypothesis, $w' \geq 0$. ■

The multi-step reduction relation is preserved, together with the length of its derivation, when multiplying both initial and final weight by the same non-negative number.

Lemma 36 *If $(M, w, s) \rightarrow^k (M', w', s')$, then for any $w^* \geq 0$, $(M, ww^*, s) \rightarrow^k (M', w'w^*, s')$*

Proof: By induction on k , with appeal to Lemma 29. ■

Similarly, adding a suffix to the trace preserves the reduction and the length of its derivation.

Lemma 37 *If $(M, w, s) \rightarrow^k (M', w', s^*)$, then for any s' , $(M, w, s@s') \rightarrow^k (M', w', s^*@s')$*

Proof: By induction on k , with appeal to Lemma 30. ■

The closure of the small-step semantics is *transitive*:

Lemma 38 *If both $(M, 1, s) \Rightarrow (M', w', \square)$ and $(M', 1, s') \Rightarrow (M'', w'', \square)$, then $(M, 1, s@s') \Rightarrow (M'', w'w'', \square)$.*

Proof: By Lemma 37, $(M, 1, s@s') \Rightarrow (M', w', s')$ and by Lemma 35, $w' \geq 0$. Hence, by Lemma 36, $(M', w', s') \Rightarrow (M'', w'w'', \square)$, which gives $(M, 1, s@s') \Rightarrow (M'', w'w'', \square)$. ■

6.2.1 Equivalence of Small-step and Big-step Semantics

In this section, we prove that the big-step and small-step semantics are equivalent and use this fact to show that the big-step semantics is deterministic. Since the big-step semantics does not use contexts explicitly, we begin by stating two auxiliary lemmas about the big-step semantics, saying that expressions reducing to `fail` also reduce to `fail` when plugged into a context.

Lemma 39 *For any E , $E[\text{fail}] \Downarrow_1^\square \text{fail}$.*

Proof: By induction on the structure of E .

- Base case: $E = []$, the result follows by (EVAL VAL).
- Induction step:
 - Case $E = (\lambda x.L) E'$: By induction hypothesis, $E'[\text{fail}] \Downarrow_1^\square \text{fail}$, and by (EVAL APPL RAISE2), $(\lambda x.L) E'[\text{fail}] \Downarrow_1^\square \text{fail}$, as required.
 - Case $E = E' L$: By induction hypothesis, $E'[\text{fail}] \Downarrow_1^\square \text{fail}$, so by (EVAL APPL RAISE1), we get $E'[\text{fail}] L \Downarrow_1^\square \text{fail}$.

■

Lemma 40 *For any closed E , if $\text{pdf}_D(\vec{c}, c) = 0$, then $E[D(\vec{c})] \Downarrow_0^{[c]} \text{fail}$.*

Proof: By induction on the structure of E .

- Base case: $E = []$, the result follows by (EVAL RANDOM FAIL).
- Induction step:
 - Case $E = (\lambda x.L) E'$: By induction hypothesis, $E'[D(\vec{c})] \Downarrow_0^{[c]} \text{fail}$, and by (EVAL APPL RAISE2), $(\lambda x.L) E'[D(\vec{c})] \Downarrow_0^\square \text{fail}$, as required.
 - Case $E = E' L$: By induction hypothesis, $E'[D(\vec{c})] \Downarrow_0^{[c]} \text{fail}$, so by (EVAL APPL RAISE1), we get $E'[D(\vec{c})] L \Downarrow_0^\square \text{fail}$.

■

The following lemma, used in the induction step in the main proof, shows how a single step of the small-step semantics can be simulated by big-step semantics.

Lemma 41 *If $(M, 1, s) \rightarrow (M', w, [])$ and $M' \Downarrow_{w'}^{s'} G$, then $M \Downarrow_{w \cdot w'}^{s @ s'} G$.*

Proof: In Appendix 6 .

■

We can now proceed to the main theorem of this chapter, stating that the closure of small-step semantics reducing an expression completely to a generalised value is equivalent to the big-step semantics.

Theorem 4 $M \Downarrow_w^s G$ if and only if $(M, 1, s) \Rightarrow (G, w, [])$.

Proof: As usual, we split the equivalence into two implications:

(1) if $M \Downarrow_w^s G$, then $(M, 1, s) \Rightarrow (G, w, [])$:

The proof is by induction on the derivation of $M \Downarrow_w^s G$.

(EVAL VAL)

• Case: $\frac{G \in \mathcal{GV}}{G \Downarrow_1^[] G}$

Here, $M = V$, $w = 1$ and $s = []$. so (M, w_0, s_0) reduces to (V, w_0, s_0) in 0 steps by the small-step semantics.

(EVAL RANDOM)

$w = \text{pdf}_D(\vec{c}, c)$

• Case: $w > 0$

$\frac{}{D(\vec{c}) \Downarrow_w^{[c]} c}$

By (RED RANDOM) (taking $E = []$), $(D(\vec{c}), 1, [c]) \rightarrow (c, w, [])$.

(EVAL RANDOM FAIL)

• Case: $\frac{\text{pdf}_D(\vec{c}, c) = 0}{D(\vec{c}) \Downarrow_0^{[c]} \text{fail}}$

By (RED RANDOM FAIL) (taking $E = []$), $(D(\vec{c}), 1, [c]) \rightarrow (\text{fail}, 0, [])$.

(EVAL PRIM)

• Case: $\frac{}{g(\vec{c}) \Downarrow_1^[] \sigma_g(\vec{c})}$

By (RED PURE) (taking $E = []$), $(g(\vec{c}), 1, []) \rightarrow (\sigma_g(\vec{c}), 1, [])$.

(EVAL SCORE)

• Case: $\frac{c \in (0, 1]}{\text{score}(c) \Downarrow_c^[] \text{true}}$

By (RED SCORE) (taking $E = []$), $(D(\vec{c}), 1, []) \rightarrow (c, w, [])$.

(EVAL APPL)

$M \Downarrow_{w_1}^{s_1} \lambda x. M'$

• Case: $\frac{N \Downarrow_{w_2}^{s_2} V \quad M' \{V/x\} \Downarrow_{w_3}^{s_3} G}{M N \Downarrow_{w_1 \cdot w_2 \cdot w_3}^{s_1 @ s_2 @ s_3} G}$

By induction hypothesis, $(M, 1, s_1) \Rightarrow (\lambda x.M', w_1, [])$, $(N, 1, s_2) \Rightarrow (V, w_2, [])$
and $(M' \{V/x\}, 1, s_3) \Rightarrow (G, w_3, [])$.

By Lemma 33 (for $E = [] N$), $(M N, 1, s_1) \Rightarrow ((\lambda x.M') N, w_1, [])$.

By Lemma 33 again (for $E = (\lambda x.M') []$), $((\lambda x.M') N, 1, s_2) \Rightarrow ((\lambda x.M') V, w_2, [])$.

By Lemma 38, $(M N, 1, s_1 @ s_2) \Rightarrow ((\lambda x.M') V, w_1 w_2, [])$

By (RED PURE), $((\lambda x.M') V, w_1 \cdot w_2, []) \rightarrow (M'[V/x], w_1 \cdot w_2, [])$, which implies $(M N, 1, s_1 @ s_2) \Rightarrow ((\lambda x.M') V, w_1 w_2, [])$

Thus, the desired result follows by Lemma 38.

(EVAL APPL RAISE1)

- Case: $\frac{M \Downarrow_w^s \text{fail}}{M N \Downarrow_w^s \text{fail}}$

By induction hypothesis, $(M, 1, s) \Rightarrow (\text{fail}, w, [])$.

By Lemma 34 (with $E = [] N$), $(M N, 1, s) \Rightarrow (\text{fail}, w, [])$.

(EVAL APPL RAISE2)

- Case: $\frac{M \Downarrow_w^s c}{M N \Downarrow_w^s \text{fail}}$

By induction hypothesis, $(M, 1, s) \Rightarrow (c, w, [])$. By Lemma 33 (with $E = [] N$), $(M N, 1, s) \Rightarrow (c N, w, [])$.

By (RED PURE), $(c N, w, []) \rightarrow (\text{fail}, w, [])$.

Thus, $(M N, 1, s) \Rightarrow (\text{fail}, w, [])$.

(EVAL APPL RAISE3)

$M \Downarrow_{w_1}^{s_1} \lambda x.M'$

- Case: $\frac{N \Downarrow_{w_2}^{s_2} \text{fail}}{M N \Downarrow_{w_1 \cdot w_2}^{s_1 @ s_2} \text{fail}}$

By induction hypothesis, $(M, 1, s_1) \Rightarrow (\lambda x.M', w_1, [])$, and $(N, 1, s_2) \Rightarrow (\text{fail}, w_2, [])$.

By Lemma 33, $(M N, 1, s_1) \Rightarrow ((\lambda x.M') N, w_1, [])$.

By Lemma 34, $((\lambda x.M') N, 1, s_2) \Rightarrow (\text{fail}, w_2, [])$.

Thus, by Lemma 38, $(M N, 1, s_1 @ s_2) \Rightarrow (\text{fail}, w_1 \cdot w_2, [])$.

(EVAL IF TRUE)

- Case: $\frac{M_2 \Downarrow_w^s G}{\text{if true then } M_2 \text{ else } M_3 \Downarrow_w^s G}$

By (RED PURE) (taking $E = []$), $(\text{if true then } M_2 \text{ else } M_3, 1, s) \rightarrow (M_2, 1, s)$.

By induction hypothesis, $(M_2, 1, s) \Rightarrow (G, w, [])$.

Hence $(\text{if 1 then } M_2 \text{ else } M_3, 1, s) \Rightarrow (G, w, [])$.

- Case (EVAL IF FALSE): analogous to (EVAL IF TRUE)

(EVAL FAIL)

- Case: $\frac{}{T \Downarrow_1^{\square} \text{fail}}$
By (RED PURE), $(T, 1, \square) \rightarrow (\text{fail}, 1, \square)$.

(2) If $(M, 1, s) \Rightarrow (G, w, \square)$ then $M \Downarrow_w^s G$:

We prove this statement by induction on the derivation of $(M, 1, s) \Rightarrow (G, w, \square)$, with appeal to Lemma 41.

- Base case: If $(M, 1, s) = (G, w, \square)$, then $M \Downarrow_s^w G$ by (EVAL VAL).
- Induction step: assume $(M, 1, s) \rightarrow (M', w', s') \rightarrow^n (G, w, \square)$. If $(M, 1, s) \rightarrow (M', w', s')$ was derived with (RED RANDOM FAIL), then $M = E[D(\vec{c})]$, $n = 1$, $s = [c]$, $G = \text{fail}$ and $w = w' = \text{pdf}_D(\vec{c}, c) = 0$. By Lemma 40, we have $M \Downarrow_0^{[c]} \text{fail}$, as required.
Otherwise, by Lemma 28, $w' > 0$, so by Lemma 36, $(M', 1, s') \rightarrow^n (G, w/w', \square)$. By induction hypothesis, $M' \Downarrow_{w/w'}^{s'} G$. By Lemma 31, $(M, 1, s^*) \rightarrow (M', w', \square)$, where $s = s^* @ s'$.
Therefore, by Lemma 41, $M \Downarrow_w^{s^* @ s'} G$, and so $M \Downarrow_w^s G$.

■

Now that we know that the big-step and small-step semantics are equivalent, and the small-step semantics is deterministic for a fixed trace, we can state that the big-step semantics is also deterministic.

Lemma 42 *If $M \Downarrow_w^s G$ and $M \Downarrow_{w'}^{s'} G'$, then $w = w'$ and $G = G'$.*

Proof: Corollary of Lemma 32 and Theorem 4. ■

6.3 A Distribution on Program Outcomes

In the previous section, we have defined the operational semantics of the probabilistic λ -calculus, which specifies the value returned by the program for each fixed random trace. However, in probabilistic modelling, we are usually interested in the *distributions* on output values, rather than just single, isolated values sampled from the model.

In this section, we use the sampling-based semantics to define a subprobability distribution on outcomes of a given program, like we did for expressions in queries in

Tabular in Chapter 4. Since the space of traces, over which we integrate the density derived from the semantics, consists of sequences of arbitrary length, we need to resort to measure theory and Lebesgue integration.

6.3.1 Distributions on Random Traces and Program Outcomes

Using the the sampling-based semantics and the measure space of program traces described in Section 3.2.2, we can now define the distributions on program traces and outcomes.

A functional view of sampling-based semantics We begin by interpreting the sampling-based semantics as a pair of (total) functions, parametrised by the given program, mapping traces to outcomes and weights. These functions are similar to those used in the random semantics of Tabular. The first of this functions, returning the outcome of evaluation, is defined as follows:

$$\mathbf{O}_M(s) \triangleq \begin{cases} G & \text{if } M \Downarrow_w^s G \text{ for some } w \in \mathbb{R}_+ \\ \text{fail} & \text{otherwise} \end{cases}$$

The second function, defining the density of a trace in the given program, is defined as:

$$\mathbf{P}_M(s) \triangleq \begin{cases} w & \text{if } M \Downarrow_w^s G \text{ for some } G \in \mathcal{GV} \\ 0 & \text{otherwise} \end{cases}$$

A σ -algebra on syntactic terms As we interpret distributions as subprobability measures, we need a measurable space on the set of programs in order to define a distribution on outcomes of evaluation. To this end, we first define a *metric* on syntactic terms as follows:

Metric on terms: $d(M, N)$

$$d(x, x) \triangleq 0$$

$$d(c, d) \triangleq |c - d|$$

$$d(M \ N, L \ P) \triangleq d(M, L) + d(N, P)$$

$$d(\lambda x.M, \lambda x.N) \triangleq d(M, N)$$

$$d(g(V_1, \dots, V_{|g|}), g(V'_1, \dots, V'_{|g|})) \triangleq d(V_1, V'_1) + \dots + d(V_{|g|}, V'_{|g|})$$

$$d(D(V_1, \dots, V_{|D|}), D(V'_1, \dots, V'_{|D|})) \triangleq d(V_1, V'_1) + \dots + d(V_{|D|}, V'_{|D|})$$

$$\begin{aligned}
d(\text{score}(V), \text{score}(W)) &\triangleq d(V, W) \\
d(\text{if } V \text{ then } M_1 \text{ else } M_2, \text{if } W \text{ then } N_1 \text{ else } N_2) &\triangleq d(V, W) + d(M_1, N_1) + d(M_2, N_2) \\
d(M, N) &\triangleq \infty \quad \text{otherwise}
\end{aligned}$$

It is easy to check that d is, indeed, a metric, and so (Λ, d) is a metric space. We define \mathcal{M} to be the Borel σ -algebra on Λ induced by the metric d . We denote by $\mathcal{M}|_{C\Lambda}$ and $\mathcal{M}|_{\mathcal{GV}}$ the restrictions of \mathcal{M} to, respectively, the set $C\Lambda$ of closed terms and the set \mathcal{GV} of generalised values (both of which are closed subsets of the metric space (Λ, d) , and so are measurable).

Distributions on traces and outcomes For every closed program M , the functions \mathbf{O}_M and \mathbf{P}_M are measurable, which we prove in the appendix.

Lemma 43 *For every closed M , the function \mathbf{O}_M is measurable $\mathcal{S} / \mathcal{M}|_{\mathcal{GV}}$.*

Proof: In Appendix E. ■

Lemma 44 *For every closed M , \mathbf{P}_M is measurable $\mathcal{S} / \mathcal{B}|_{\mathbb{R}_+}$.*

Proof: In Appendix E. ■

The distribution $\langle\langle M \rangle\rangle$ on program traces, applied to a set of traces $B \in \mathcal{S}$ is simply the integral of the density \mathbf{P}_M with respect to μ restricted to B . Recall that μ is the stock measure on program traces defined in Section 3.2.2.

$$\langle\langle M \rangle\rangle(B) \triangleq \int_B \mathbf{P}_M(s) \mu(ds) = \int_B \mathbf{P}_M(s) ds$$

Lemma 45 *$\langle\langle M \rangle\rangle$ is a subprobability measure on $(\mathbb{U}, \mathcal{S})$.*

Proof: In Appendix E. ■

From the trace distribution, we can obtain the distribution on output values $\llbracket M \rrbracket_{\mathbb{U}}$ by transforming $\langle\langle M \rangle\rangle$ by the result function \mathbf{O}_M .

$$\llbracket M \rrbracket_{\mathbb{U}} \triangleq \langle\langle M \rangle\rangle \mathbf{O}_M^{-1}$$

By expanding this definition, we can write $\llbracket M \rrbracket_{\mathbb{U}}(A)$ as follows:

$$\llbracket M \rrbracket_{\mathbb{U}}(A) = \langle\langle M \rangle\rangle(\mathbf{O}_M^{-1}(A)) = \int_{\mathbf{O}_M^{-1}(A)} \mathbf{P}_M(s) ds = \int \mathbf{P}_M(s) [\mathbf{O}_M(s) \in A] ds$$

Intuitively, $\mathbf{O}_M^{-1}(A)$ is the set of traces which yield an output value in A in the program M , so $\llbracket M \rrbracket_{\mathbb{U}}(A)$ is the integral of the density over just the set of traces for which the output value is in A .

Theorem 5 $\llbracket M \rrbracket_{\mathbb{U}}$ is a subprobability measure on $(\mathcal{GV}, \mathcal{M}|_{\mathcal{GV}})$.

Proof: In Appendix E. ■

The measure $\llbracket M \rrbracket_{\mathbb{U}}$ defines a distribution on *generalised* values, treating the exception `fail` like a value. We can also define a distribution $\llbracket M \rrbracket_{\mathbb{U}}|_{\mathcal{V}}$ on values, excluding `fail`, via a restricted distribution on traces $\langle\langle M \rangle\rangle^{\mathcal{V}}$ and a restricted density function $\mathbf{P}_M^{\mathcal{V}}$ (which will also be useful in Chapter 7).

$$\mathbf{P}_M^{\mathcal{V}}(s) \triangleq \begin{cases} w & \text{if } M \Downarrow_w^s V \text{ for some } V \in \mathcal{V} \\ 0 & \text{otherwise} \end{cases}$$

Lemma 46 For every closed M , $\mathbf{P}_M^{\mathcal{V}}$ is measurable $\mathcal{S}/\mathcal{B}|_{\mathbb{R}_+}$.

Proof: In Appendix E. ■

$$\langle\langle M \rangle\rangle^{\mathcal{V}}(B) \triangleq \int_B \mathbf{P}_M^{\mathcal{V}}(s) ds = \langle\langle M \rangle\rangle(B \cap \mathbf{O}_M^{-1}(\mathcal{V}))$$

Lemma 47 For every closed M , $\langle\langle M \rangle\rangle^{\mathcal{V}}$ is a subprobability measure on $(\mathbb{U}, \mathcal{S})$.

Proof: In Appendix E. ■

$$\llbracket M \rrbracket_{\mathbb{U}}|_{\mathcal{V}}(A) \triangleq \langle\langle M \rangle\rangle^{\mathcal{V}}(\mathbf{O}_M^{-1}(A)) = \int \mathbf{P}_M^{\mathcal{V}}(s) [\mathbf{O}_M(s) \in A] ds$$

Lemma 48 For every closed M , $\llbracket M \rrbracket_{\mathbb{U}}|_{\mathcal{V}}$ is a measure on $(\mathcal{GV}, \mathcal{M}|_{\mathcal{GV}})$.

Proof: In Appendix E. ■

Note that because of the kind of measure restriction we are using, $\llbracket M \rrbracket_{\mathbb{U}}|_{\mathcal{V}}$ is defined on generalised values and $\llbracket M \rrbracket_{\mathbb{U}}|_{\mathcal{V}}(\{\text{fail}\}) = 0$. Alternatively, we could have defined $\llbracket M \rrbracket_{\mathbb{U}}|_{\mathcal{V}}$ to be a measure on $(\mathcal{V}, \mathcal{M}|_{\mathcal{V}})$.

Example: geometric distribution Let us demonstrate the calculation of distributions by revisiting the geometric distribution example from section 6.1.3:

```

let geometric =
  (fix g.
    λp. (let z = rnd() in
      let y = (z < p) in
      if y then 0 else 1 + (g p))) in
let n = geometric 0.5 in
let b = n > 1 in
if b then n else fail

```

Let $S_n = \{s' @ [c] \mid s' \in [0.5, 1]^n, c \in [0, 0.5)\}$. It is easy to check that $\mathbf{O}_{\text{geometric } 0.5}([]) = \text{fail}$, $\mathbf{P}_{\text{geometric } 0.5}([]) = 0$ and for any s of length $n + 1$, we have:

$$\mathbf{O}_{\text{geometric } 0.5}(s) = \begin{cases} n & \text{if } n > 1 \text{ and } s \in S_n \\ \text{fail} & \text{otherwise} \end{cases}$$

and

$$\mathbf{P}_{\text{geometric } 0.5}(s) = [s \in S_n]$$

Thus, we have:

$$\begin{aligned}
\langle\langle \text{geometric } 0.5 \rangle\rangle(B) &= \int_B \mathbf{P}_{\text{geometric } 0.5}(s) ds \\
&= \sum_{n=0}^{\infty} \int_{\mathbb{R}^{n+1} \cap B} \mathbf{P}_{\text{geometric } 0.5}(s) \lambda_{n+1}(ds) \\
&= \sum_{n=0}^{\infty} \int_{\mathbb{R}^{n+1} \cap B} [s \in S_n] \lambda_{n+1}(ds) \\
&= \sum_{n=0}^{\infty} \lambda_{n+1}(B \cap S_n)
\end{aligned}$$

This leads to the value distribution assigning probabilities to individual outcomes $n > 1$ as follows:

$$\begin{aligned}
\llbracket \text{geometric } 0.5 \rrbracket_{\mathbb{U}}(\{n\}) &= \langle\langle \text{geometric } 0.5 \rangle\rangle(\mathbf{O}_{\text{geometric } 0.5}^{-1}\{n\}) \\
&= \langle\langle \text{geometric } 0.5 \rangle\rangle(S_n) \\
&= \lambda_{n+1}(S_n) \\
&= \frac{1}{2^{n+1}}
\end{aligned}$$

In case $n \leq 1$, we have $\mathbf{O}_{\text{geometric } 0.5}^{-1}(\{n\}) = \emptyset$, so $\llbracket \text{geometric } 0.5 \rrbracket_{\mathbb{U}}(\{n\}) = 0$. Moreover, the probability of the observation $n > 1$ not being satisfied is:

$$\begin{aligned} \llbracket \text{geometric } 0.5 \rrbracket_{\mathbb{U}}(\{\text{fail}\}) &= \langle\langle \text{geometric } 0.5 \rangle\rangle \left(S_0 \cup S_1 \cup \left(\bigcup_{i=2}^{\infty} \mathbb{R}^{i+1} \setminus S_i \right) \right) \\ &= \langle\langle \text{geometric } 0.5 \rangle\rangle (S_0 \cup S_1) \\ &= \lambda_1(S_0) + \lambda_2(S_1) \\ &= \frac{3}{4} \end{aligned}$$

where the second equality follows from the fact that $\mathbf{P}_{\text{geometric } 0.5}(s) = 0$ for all traces s under which the program `geometric 0.5` does not evaluate to a generalised value. Note that $\llbracket \text{geometric } 0.5 \rrbracket_{\mathbb{U}}$ is a probability measure due to the program terminating with probability 1 and containing no calls to `score`, but in general, for an arbitrary program M , $\llbracket M \rrbracket_{\mathbb{U}}$ only defines a subprobability measure.

Now, let us derive the distributions on traces and values restricted to successful outcomes. We have

$$\mathbf{P}_{\text{geometric } 0.5}^{\mathcal{V}}(s) = \begin{cases} 1 & \text{if } n > 1 \text{ and } s \in S_n \\ 0 & \text{otherwise} \end{cases}$$

Thus:

$$\begin{aligned} \langle\langle \text{geometric } 0.5 \rangle\rangle^{\mathcal{V}}(B) &= \int_B \mathbf{P}_{\text{geometric } 0.5}^{\mathcal{V}}(s) ds \\ &= \sum_{n=2}^{\infty} \lambda_{n+1}(B \cap S_n) \end{aligned}$$

The unnormalised probability of each outcome $n > 1$ is still the same as before:

$$\llbracket \text{geometric } 0.5 \rrbracket_{\mathbb{U}|\mathcal{V}}(\{n\}) = \langle\langle \text{geometric } 0.5 \rangle\rangle^{\mathcal{V}}(\mathbf{O}_{\text{geometric } 0.5}^{-1}\{n\}) = \frac{1}{2^{n+1}}$$

Similarly, $\llbracket \text{geometric } 0.5 \rrbracket_{\mathbb{U}|\mathcal{V}}(\{n\}) = 0$ for $n \leq 1$. The difference is that $\llbracket \text{geometric } 0.5 \rrbracket_{\mathbb{U}|\mathcal{V}}$ is restricted to values, so $\llbracket \text{geometric } 0.5 \rrbracket_{\mathbb{U}|\mathcal{V}}(\{\text{fail}\}) = 0$. This affects the normalisation: since we have $\llbracket \text{geometric } 0.5 \rrbracket_{\mathbb{U}|\mathcal{V}}(\mathcal{G}\mathcal{V}) = \llbracket \text{geometric } 0.5 \rrbracket_{\mathbb{U}|\mathcal{V}}(\mathcal{V}) = \frac{1}{4}$, the normalised probability of each outcome $n > 1$ is now $\frac{1}{2^{n-1}}$.

Discrete variables For simplicity, we have decided to only include continuous distributions in the language and to restrict the space of traces to sequences of reals.

Discrete random draws can be simulated by continuous ones, like flip in the geometric distribution example. For a less trivial example of such encoding, consider the Discrete distribution (as shown in Chapter 4) with a n -dimensional parameter vector $[p_1, \dots, p_n]$. A draw from this distribution could be simulated by drawing a value c from the uniform distribution rnd on the unit interval and then returning k such that $p_1 + \dots + p_k < c(p_1 + \dots + p_n) \leq p_1 + \dots + p_{k+1}$.

The semantics of the language could, however, be easily extended to support discrete distributions. For example, we could change the space of traces \mathbb{U} to $\biguplus_{n \in \mathbb{N}} (\mathbb{R} \uplus \mathbb{N})^n$. The new σ -algebra \mathcal{S}' would then be generated by sets of the form $H^1 \times \dots \times H^n$ such that for each i , $H^i = H_{\mathbb{R}}^i \uplus H_{\mathbb{N}}^i$, where $H_{\mathbb{R}}^i \in \mathcal{B}$ and $H_{\mathbb{N}}^i \in \mathcal{P}(\mathbb{N})$.

We would also need to update μ to

$$\mu' \triangleq \sum_{n \in \mathbb{N}} (\lambda \oplus \mu_{\#})^n$$

where $\mu_{\#}$ is the counting measure on \mathbb{N} and \oplus is a disjoint sum of measures (i.e. $\lambda \oplus \mu_{\#}(H) = \lambda(H \cap \mathbb{R}) + \mu_{\#}(H \cap \mathbb{N})$).

We could then add discrete distributions to the language. In the sampling-based semantics, evaluating a discrete random draw would multiply the weight of the trace by the *probability mass function* (that is, the density of the discrete distribution with respect to the counting measure) at the given point. This could be formalized by the following new rule in the big-step semantics, which also assumes the language has integer constants n (alternatively, we could encode integers by Church numerals or use a mapping from integers to reals).

$$\begin{array}{c} \text{(EVAL RANDOM DISCRETE)} \\ \frac{w = \text{pmf}_{D_{\text{discr}}}(\vec{c}, n) \quad w > 0}{D_{\text{discr}}(\vec{c}) \Downarrow_w^{[n]} n} \end{array}$$

6.3.2 Digression: Motivation for Bounded Scores

We have restricted `score` to only accept arguments in $(0, 1]$ to ensure that the trace distributions $\langle\langle M \rangle\rangle$ and $\langle\langle M \rangle\rangle^{\mathcal{V}}$ are subprobability measures. Indeed, because of recursion, if we allowed unbounded scores, the measure $\langle\langle M \rangle\rangle$ could be non-finite or even non- σ -finite, even if M terminated with probability 1. For example, consider the following program *inflate*:

$$\text{inflate} \triangleq \text{fix } f \ (\lambda x. \text{if flip}(0.5) \text{ then } (\text{score}(2); f\ x) \text{ else } x)$$

For every value $V \in \mathcal{V}$, the program *inflate* V terminates with probability 1, because the probability of not returning the value V after n calls to `flip` is $\frac{1}{2^n}$, which goes to 0 as n goes to infinity. However, it is easy to check that $\langle\langle \textit{inflate } V \rangle\rangle(\mathbb{U}) = \infty$, and that $\llbracket \textit{inflate } V \rrbracket_{\mathbb{U}}(A) = \infty$ if $V \in A$ and 0 otherwise. It is, then, impossible to normalize $\langle\langle \textit{inflate } V \rangle\rangle$ to obtain a probability measure on traces.

Alternatively, we could have allowed unbounded scores and only considered programs M such that $\llbracket M \rrbracket_{\mathbb{U}}|_{\mathcal{V}} < \infty$. We decided to use bounded scores instead, to avoid having to add this condition to all lemmas in the next chapter.

6.4 Conclusions

In this chapter, we have defined an operational sampling-based semantics of an untyped lambda-calculus with continuous random draws and soft and hard conditioning. Our semantics reduces a program to an output value (or the exception `fail`) and a weight, given a particular, fixed random trace. We define the distribution on output values of a program as an integral of the weight with respect to a generalisation of the Lebesgue measure to the space of program traces.

To our best knowledge, this is the first semantics of a Turing-complete functional probabilistic language with continuous random draws and conditioning which defines a probability distribution on output values of arbitrary programs. Admittedly, these values are treated syntactically, which means we do not define distributions on lambda terms treated as mathematical functions.

This semantics forms the basis of Chapter 7, where we prove correctness of Metropolis-Hastings inference in functional probabilistic programs by showing that the distribution of samples converges to the program semantics.

Individual Contributions

The vast majority of the work on the semantics of the probabilistic lambda-calculus presented here is my work, done with help from other authors on the paper this chapter is based on [Borgström et al., 2016]. All the proofs presented in this chapter, as well as Appendix E, are my own work¹.

¹With the exception of the short Lemma 138, proven by Johannes Borgström

Chapter 7

Correct Metropolis-Hastings for Functional Probabilistic Programs

Acknowledgement This chapter is based on the paper “A Lambda Calculus Foundation for Universal Probabilistic Programming” [Borgström et al., 2016] published at the 2016 International Conference on Functional Programming (ICFP). The paper was joint work with Johannes Borgström, Ugo Dal Lago and Andrew D. Gordon.

Inference in probabilistic languages such as Tabular, in which programs correspond directly to factor graphs, can be performed by using various approximate message-passing algorithms, such as Expectation Propagation [Minka, 2001] used by Infer.NET [Winn and Minka, 2009], the default backend of Tabular.

Because of the presence of recursion, and thus the number of random variables possibly changing across runs, these inference techniques for fixed-dimensional models do not apply to languages such as Church, discussed in Chapter 6. Instead, the algorithm proposed by Goodman et al. [2008] and described in more detail in [Wingate et al., 2011] (which we hereafter call Trace MCMC), is a version of the Metropolis-Hastings algorithm [Metropolis et al., 1953, Hastings, 1970] which constructs a Markov chain on the space of program traces—that is, lists of random variables sampled during the execution of the program. At each step, the algorithm proposes some change to the last accepted trace, and accepts or rejects the new trace with some probability depending on the acceptance ratio, as in usual Metropolis-Hastings.

The Trace MCMC algorithm lacked any formal proof of correctness and bugs have, indeed, been found in Wingate’s formulation [Kiselyov, 2016, Cai, 2016]. Verifying

correctness of inference algorithms is important in times when probabilistic programming is increasingly being used in safety-critical settings, such as nuclear test detection [Arora, 2011], road detection for autonomous vehicles [Mansinghka et al., 2013] and Wide Area Motion Imagery (a recent DARPA PPAML challenge problem), which can be used for tracking movements of an enemy army.

This chapter presents the first formal proof of correctness of a variant of Trace MCMC for a functional probabilistic language. This algorithm is inspired by, but slightly different from the one presented by Wingate et al. [2011]—the differences between the two are discussed in Section 7.5.1.

We define this variant of Trace MCMC for the core calculus presented in Chapter 6 and formalize the algorithm, defining the proposal and transition kernels by integrating the transition kernel density. We then leverage results from literature on MCMC on generalized state spaces [Tierney, 1994, Roberts et al., 2004] to show that the distribution of samples returned by the algorithm converges to the value distribution of the program (defined by the semantics). Our proof is rigorous and includes proving measurability of the function mapping traces to weights, proving measurability of the proposal kernel and showing that the transition kernel is indeed a probability kernel.

The main technical contribution of this chapter is a formal proof of correctness of a variant of Trace MCMC for a Turing-complete functional probabilistic language.

7.1 A Metropolis-Hastings Sampling Algorithm

We start by defining our version of the Metropolis-Hastings [Metropolis et al., 1953, Hastings, 1970] inference algorithm, in a generative way. The algorithm is defined on the sample space of program traces (i.e. finite lists of real values of arbitrary lengths), which is the measurable space defined in Section 3.2.2. Hence, each random variable in the Markov Chain is trace-valued.

Recall that $(M, w, s) \rightarrow (M', w', s')$ means that the program M with initial weight w reduces to M' with the random trace s in one step, where w' is the updated weight after performing the reduction and s' is the unused suffix of trace s . We write $(M, w, s) \Rightarrow (M', w', s')$ for the reflexive and transitive closure of the above relation—that is, $(M, w, s) \Rightarrow (M', w', s')$ means that M with initial weight w reduces to M' in zero or more steps, where, again, w' is the updated weight and s' the unused suffix of s . The function $\mathbf{P}_M^\mathcal{V}$ is the *density* of the program M —that is, $\mathbf{P}_M^\mathcal{V}(s)$ returns the weight of the trace s , which is set to 0 if the trace does not lead to a value. More formally, $\mathbf{P}_M^\mathcal{V}(s) = w$

if $(M, 1, s) \Rightarrow (V, w, [])$ for some value V and $\mathbf{P}_M^{\mathcal{V}}(s) = 0$ otherwise.

The algorithm we are constructing aims to approximate the distribution on traces $\langle\langle M \rangle\rangle^{\mathcal{V}}(B) = \int_B \mathbf{P}_M^{\mathcal{V}}(s) ds$, so $\mathbf{P}_M^{\mathcal{V}}$ is the density of the target distribution.

Assuming that the proposal kernel of the algorithm has a *density* $q(s, t)$, from which we can sample a new trace t given a previous trace s , the algorithm has the following form.

- (1) Sample an initial trace s such that $\mathbf{P}_M^{\mathcal{V}}(s) \neq 0$ (for example, by performing rejection sampling until a valid trace is obtained)
- (2) Propose a new trace t from the density $q(s, \cdot)$
- (3) Accept the trace t with probability

$$\alpha(s, t) = \min \left(1, \frac{\mathbf{P}_M^{\mathcal{V}}(t)q(t, s)}{\mathbf{P}_M^{\mathcal{V}}(s)q(s, t)} \right)$$

- (4) If the trace is accepted, output trace t , set $s := t$ and repeat from step 2. Otherwise, output the old trace s and also repeat from 2.

The above generic algorithm is parametric on the proposal density $q(s, t)$, which we take to be the density corresponding to the following proposal procedure: Suppose that $s = [c_1, \dots, c_n]$ is a valid trace. A new trace t is proposed as follows:

- (1) Let $k = 1$ and $w = 1$ and $t = []$.
- (2) Let N be the expression obtained by reducing M deterministically as long as possible—that is, $(M, w, []) \Rightarrow (N, w', [])$ and N is not a deterministic redex or $\text{score}(V)$ in context. If $N \in \mathcal{V}$, return t with weight w' . If $N = \text{fail}$, return $[]$ with weight w' .
- (3) Otherwise, if $N = E[D(\vec{c})]$, then:
 - If $k \leq n$, we sample d_k from a Gaussian centred at c_k (with some fixed variance σ^2).
 - If $k > n$, we sample d_k from the target distribution $D(\vec{c})$.

Let N' , w'' be such that $(N, w, [d_k]) \rightarrow (N', w'', [])$. Set $M = N'$, $w = w''$ and $t = t @ [d_k]$ and $k := k + 1$ and repeat from step 2.

Informally, the new trace t is obtained by evaluating the expression M such that when the i -th random draw is reached, instead of sampling from the target distribution, we sample from a Gaussian centred at s_i (if it exists), and we only start sampling from the target when we run out of random values in the previous trace s .

We chose to always use Gaussian proposals in the algorithm because the unbounded support of the Gaussian distribution helps to ensure ϕ -irreducibility (formally proven later in this chapter), which we use to prove convergence. The downside of this choice is that using a Gaussian to propose a new value for a distribution with bounded support (for example the Gamma distribution, which is only defined on positive real numbers) may result in the new value being outside the support of this target distribution. This leads to the new trace being rejected. Note that this affects the performance of the algorithm, but not its formal correctness.

Density of the proposal We now need to define the density of the above proposal. Since the proposal process relies on reducing an already partially evaluated expression, it is convenient to define an auxiliary function $\text{peval}(M, s)$, which reduces the closed term M with (possibly incomplete) trace s and returns the expression obtained after fully consuming s . Formally, this function is defined as follows:

$$\text{peval}(M, s) \triangleq \begin{cases} M & \text{if } s = [] \\ M' & \text{if } (M, 1, s) \Rightarrow (\hat{M}, \hat{w}, \hat{s}) \rightarrow (M', w', []) \\ & \text{for some } \hat{M}, \hat{w}, \hat{s}, w' \text{ such that } \hat{s} \neq [] \\ \text{fail} & \text{otherwise} \end{cases}$$

Lemma 49 *The function peval is measurable $\mathcal{M}|_{C\Lambda} \times \mathcal{S} / \mathcal{M}|_{C\Lambda}$*

The peval operator satisfies the following important property:

Lemma 50 *For all closed M, s, t , $\text{peval}(\text{peval}(M, s), t) = \text{peval}(M, s @ t)$*

The density $q(s, t)$ of a valid trace t is then the product of the densities of the elements of s obtained by perturbing the elements of t , that is, $\prod_{i=1}^k \text{pdf}_{\text{Gaussian}}(s_i, \sigma^2, t_i)$, where $k = \min\{|s|, |t|\}$, multiplied by the product of densities of the target distributions in $N = \text{peval}(M, s)$ from which the remaining elements were sampled, in case $|t| > |s|$. It might be tempting to define the latter product as $\mathbf{P}_N^{\mathcal{V}}([t_{k+1}, \dots, t_{|t|}])$ —however, the weight calculated this way would take into account calls to `score` reached

after we ran out of the previous trace s . Because of this, $q(s, t)$ would not match the actual density of the proposal procedure outlined above, which ignores soft conditioning. Besides, implementing a proposal procedure matching this naive density $q(s, t) = \prod_{i=1}^k \text{pdf}_{\text{Gaussian}}(s_i, \sigma^2, t_i) \mathbf{P}_N^{\mathcal{V}}([t_{k+1}, \dots, t_{|t|}])$ would be impractical, because the Metropolis-Hastings algorithm is supposed to propose just a trace, not a weighted trace, so accounting for the uses of score would require using an inefficient technique such as rejection sampling, to ensure that a trace is proposed from the above density.

In order to define the true density of the proposal, we first define an alternative version of $\mathbf{P}_M^{\mathcal{V}}$, which ignores soft conditioning and returns only the product of densities of distributions sampled from while evaluating M . This new function will itself be defined in terms of a modified sampling-based semantics. We let the judgment $M \Downarrow_w^s G$ be defined inductively by the same set of rules as $M \Downarrow_w^s G$, but with (EVAL SCORE) replaced with:

$$\begin{array}{c} \text{(EVAL SCORE SKIP)} \\ c \in (0, 1] \\ \hline \text{score}(c) \Downarrow_1 \text{true} \end{array}$$

For completeness, we also define a small-step version of this modified semantics, $(M, w, s) \rightsquigarrow (M', w', s')$, which is defined inductively by the same set of rules as $(M, w, s) \rightarrow (M', w', s')$ but with (RED SCORE) replaced with:

$$\begin{array}{c} \text{(RED SCORE SKIP)} \\ c \in (0, 1] \\ \hline (E[\text{score}(c)], w, s) \rightsquigarrow (E[\text{true}], w, s) \end{array}$$

We let \rightsquigarrow^* be the reflexive and transitive closure of \rightsquigarrow .

We can now define the desired function, \mathbf{P}_M^* , as follows:

$$\mathbf{P}_M^*(s) \triangleq \begin{cases} w & \text{if } M \Downarrow_w^s V \text{ for some } V \in \mathcal{V} \\ 0 & \text{otherwise} \end{cases}$$

All the properties of the relations \Downarrow and \rightarrow obviously hold also for \downarrow and \rightsquigarrow .

Lemma 51 *For every closed M , \mathbf{P}_M^* is measurable $\mathcal{S}/\mathcal{B}|_{\mathbb{R}_+}$*

Proof: Almost identical to the proof of measurability of $\mathbf{P}_M^{\mathcal{V}}$. ■

Using the functions `peval` and \mathbf{P}_M^* , the density $q(s, t)$ for $t \neq []$ can be defined as:

$$q(s, t) \triangleq (\prod_{i=1}^k \text{pdf}_{\text{Gaussian}}(s_i, \sigma^2, t_i)) \mathbf{P}_N^*([t_{k+1}, \dots, t_{|t|}]) \quad \text{if } t \neq []$$

where $k = \min\{|s|, |t|\}$ and $N = \text{peval}(M, [t_1, \dots, t_k])$.

The product $\prod_{i=1}^k \text{pdf}_{\text{Gaussian}}(s_i, \sigma^2, t_i)$ is the density of the part of the trace t obtained by perturbing elements of s and $\mathbf{P}_N^*([t_{k+1}, \dots, t_{|t|}])$ is the density of the suffix of the trace resampled from the target distributions in the program. Note that if $|t| \leq |s|$, the suffix $[t_{k+1}, \dots, t_{|t|}]$ is empty and N must evaluate to a generalised value without sampling any random variables for the density to be non-zero.

The density $q(s, t)$ is zero for complete traces $t \neq []$ which lead to failing observations, because the algorithm returns an empty trace in case the evaluation fails.

7.2 Transition Kernel

In order to reason about the above algorithm (applied to a fixed closed term M) using the standard measure-theoretic framework, we need to define its *transition kernel* $P(s, B)$ which for every $s \in \mathbb{U}$ and $B \in \mathcal{S}$ gives the probability that the next sampled value will be in B if the current trace is s . The transition kernel is itself defined in terms of a *proposal kernel* $Q(s, B)$, defining the probability of the next *proposed* value being in B , and the acceptance ratio $\alpha(s, t)$.

To simplify the notation, we assume we are applying the algorithm to a fixed closed term M , on which all functions defined in this section are implicitly parametric. Furthermore, to avoid dealing with degenerate cases, we assume that M is not deterministic, i.e. $\langle\langle M \rangle\rangle(\{\emptyset\}) = 0$, and that M evaluates to a value with non-zero probability, i.e. $\llbracket M \rrbracket_{\mathbb{U}}(\mathcal{V}) > 0$.

The Proposal Kernel The proposal kernel of the algorithm is the Lebesgue integral of the density $q(s, t)$ (treated as a function of t for a fixed s) with respect to the stock measure on traces. We have already defined $q(s, t)$ for non-empty traces t — to ensure that $Q(s, B)$ is a probability kernel, corresponding to the proposal procedure, we define $q(s, [])$ to be:

$$q(s, []) \triangleq 1 - \int_{\mathbb{U} \setminus \{\emptyset\}} q(s, t) dt$$

The density at \square is the probability that the proposal procedure will return an invalid trace. The density q is a measurable non-negative function, which can be Lebesgue integrated in the usual way.

Lemma 52 *For every $s, t \in \mathbb{U}$, $q(s, t) \geq 0$.*

Proof: In Appendix E. ■

Lemma 53 *The function q is measurable $\mathcal{S} \times \mathcal{S} / \mathcal{R} |_{\mathbb{R}_+}$.*

Proof: In Appendix E. ■

We can now define the proposal kernel as:

$$Q(s, B) \triangleq \int_B q(s, t) dt$$

This proposal kernel is a valid probability kernel.

Lemma 54 *Q is a probability kernel on $(\mathbb{U}, \mathcal{S})$.*

Proof: In Appendix E. ■

Transition Kernel We now define the *transition kernel* $P(s, B)$, which gives the probability of the next trace returned by the algorithm being in B if the current trace is s . The transition kernel depends on the proposal kernel Q defined above and the acceptance ratio α defined in 7.1, For completeness, we extend $\alpha(s, t)$ to the degenerate case $\mathbf{P}_M^{\mathcal{V}}(s)q(s, t) = 0$:

$$\alpha(s, t) \triangleq \begin{cases} 1 & \text{if } \mathbf{P}_M^{\mathcal{V}}(s)q(s, t) = 0 \\ \min \left\{ 1, \frac{\mathbf{P}_M^{\mathcal{V}}(t)q(t, s)}{\mathbf{P}_M^{\mathcal{V}}(s)q(s, t)} \right\} & \text{otherwise} \end{cases}$$

The transition kernel is defined in the usual way, following the literature on MCMC on generalised state spaces [Tierney, 1994].

$$P(s, B) \triangleq \int_B \alpha(s, t) Q(s, dt) + [s \in B] \int (1 - \alpha(s, t)) Q(s, dt)$$

The first term $\int_B \alpha(s, t) Q(s, dt)$ is the probability of a trace in B being proposed and accepted. The second summand adds the probability of a new trace being rejected, which also yields a trace in B if $s \in B$.

We define the n -fold closure $P^n(s, B)$ of the transition kernel P at s to be the probability that the n -th trace returned by the algorithm will be in B if s is the initial trace.

$$\begin{aligned} P^0(s, B) &\triangleq [s \in B] \\ P^{n+1}(s, B) &\triangleq \int P^n(t, B) P(s, dt) \end{aligned}$$

By applying a measure transformation to $P^n(s, B)$ we can define the probability of sampling a value in a set $A \in \mathcal{M}|_{\mathcal{V}}$ in the n -th step:

$$T^n(s, A) \triangleq P^n(s, \mathbf{O}_M^{-1}(A))$$

7.3 Correctness of Inference

Having fully defined the inference algorithm and the corresponding transition kernel, we can finally prove the main result of this section. We start by stating the main theorem, to show precisely what we mean by a sampling algorithm being correct. Below, we write $\widehat{\llbracket M \rrbracket_{\mathbb{U}}|_{\mathcal{V}}}$ for the normalised distribution $\llbracket M \rrbracket_{\mathbb{U}}|_{\mathcal{V}}$, i.e. $\widehat{\llbracket M \rrbracket_{\mathbb{U}}|_{\mathcal{V}}}(A) = \frac{\llbracket M \rrbracket_{\mathbb{U}}|_{\mathcal{V}}(A)}{\llbracket M \rrbracket_{\mathbb{U}}|_{\mathcal{V}}(\mathcal{GV})}$.

Theorem 6 *For every $s \in \mathbb{U}$,*

$$\lim_{n \rightarrow \infty} \|T^n(s, \cdot) - \widehat{\llbracket M \rrbracket_{\mathbb{U}}|_{\mathcal{V}}}\| = 0$$

Thus, a sampling-based inference algorithm is correct if the distribution -of sampled values approaches the true normalised distribution of output values of the given program (as defined by the semantics) as the number of steps goes to infinity. The remainder of this section is the proof of the above theorem, stating that Trace MCMC is a correct algorithm according to this criterion.

Our proof is based on the following known results from literature on statistics:

Lemma 55 (Roberts et al. [2004], Propositions 1 and 2) *If P is a Metropolis-Hastings kernel (as defined above) with a proposal kernel $Q(s, B) = \int_B q(s, t) dt$ and acceptance ratio $\alpha(s, t) = \min \left\{ 1, \frac{\tilde{\pi}(t)q(t, s)}{\tilde{\pi}(s)q(s, t)} \right\}$ if $\tilde{\pi}(s)q(s, t) > 0$ and $\alpha(s, t) = 0$ otherwise and $\pi(A) = \frac{\int_A \tilde{\pi}(s) ds}{\int \tilde{\pi}(s) ds}$, then π is the stationary distribution of the Markov chain defined by P .*

Lemma 56 (Roberts et al. [2004], Theorem 4 and subsequent remarks) *Let \mathcal{P} be a Markov chain on a measurable space (E, \mathcal{E}) , where \mathcal{E} is countably generated. If π is the stationary distribution of \mathcal{P} and \mathcal{P} is ϕ -irreducible and aperiodic, then for π -almost all $x \in E$,*

$$\lim_{n \rightarrow \infty} \|\mathcal{P}^n(x, \cdot) - \pi\| = 0$$

Moreover, if \mathcal{P} is Harris recurrent, the above holds for every $x \in E$.

Lemma 57 (Roberts and Rosenthal [2006], Theorem 6(vi)) *Let \mathcal{P} be a ϕ -irreducible Markov chain on (E, \mathcal{E}) with a stationary distribution π . If for every $s \in E$ and $B \in \mathcal{E}$ such that $\pi(B) = 0$, the probability of every state of \mathcal{P} being in B is 0 (that is, if $\lim_{n \rightarrow \infty} \mathcal{P}^n(s, B^n) = 0$), then \mathcal{P} is Harris-recurrent.*

We begin by showing that the Markov chain on traces defined by P converges to the normalised trace distribution $\pi(B) \triangleq \frac{\langle\langle M \rangle\rangle^{\mathcal{Y}}(B)}{\langle\langle M \rangle\rangle^{\mathcal{Y}}(\mathbb{U})} = \frac{\int_B \mathbf{P}_M^{\mathcal{Y}}(s) ds}{\int \mathbf{P}_M^{\mathcal{Y}}(s) ds}$, that is $\lim_{n \rightarrow \infty} \|P^n(x, \cdot) - \pi\| = 0$. By the above three results, we only need to prove that P is π -irreducible and aperiodic and that the probability of P staying forever in a null set is 0.

7.3.1 Additional properties of reduction

In order to prove the convergence of the Markov chain, we need some additional technical lemmas about the various reduction relations used in this chapter.

We begin by showing that the density (restricted to traces yielding values) of a closed program M at a trace s is 0 if and only if the density of M partially evaluated with an prefix of s is 0 at the corresponding suffix.

Lemma 58 $\mathbf{P}_{\text{peval}(M,s)}^{\mathcal{Y}}(t) = 0$ if and only if $\mathbf{P}_M^{\mathcal{Y}}(s@t) = 0$.

To this end, we need some auxiliary results:

Lemma 59 *If $(M, w, s@t) \rightarrow (M', w', s'@t)$, then $(M, w, s) \rightarrow (M', w', s')$*

Proof: By case analysis. ■

Lemma 60 *If $(M, 1, s@t) \Rightarrow (V, w, \square)$ and $w > 0$, then either $s = \square$ or there exist unique $M_k, w_k, s_k \neq \square, M', w'$ such that $(M, 1, s) \Rightarrow (M_k, w_k, s_k) \rightarrow (M', w', \square)$ and $(M', 1, t) \Rightarrow (V, w'', \square)$ for some $w'' > 0$.*

Proof: By induction on the length of derivation of $(M, 1, s@t) \Rightarrow (V, w, [])$:

- Base case: $(M, 1, s@t) = (V, w, [])$. We have $s = []$, as required.
- Induction step: The result is trivial if $s = []$. Now, let us assume $s \neq []$. We have $(M, 1, s@t) \rightarrow (\hat{M}, \hat{w}, \hat{s}@t) \rightarrow^k (V, w, [])$ for some \hat{M} , \hat{w} , \hat{s} and $k \geq 0$. If $(M, 1, s@t) \rightarrow (\hat{M}, \hat{w}, \hat{s}@t)$ was derived with (RED RANDOM FAIL), then $\hat{M} = E[\text{fail}]$ for some E , which is a contradiction, since $E[\text{fail}]$ can only reduce to $\text{fail} \notin \mathcal{V}$. Hence, $(M, 1, s@t) \rightarrow (\hat{M}, \hat{w}, \hat{s}@t)$ was not derived by (RED RANDOM FAIL), so by Lemma 28, $\hat{w} > 0$.
 - If $\hat{s} = []$, then by Lemma 59 we have $(M, 1, s) \rightarrow (\hat{M}, \hat{w}, [])$, so $(M, 1, s) \Rightarrow (M_k, w_k, s_k) \rightarrow (\hat{M}, \hat{w}, [])$ for $(M_k, w_k, s_k) = (M, 1, s)$. By Lemma 36, we have $(\hat{M}, 1, t) \rightarrow^k (V, w/\hat{w}, [])$, where obviously $w/\hat{w} > 0$.
 - If $\hat{s} \neq []$, then by Lemma 36, $(\hat{M}, 1, \hat{s}@t) \rightarrow^k (V, w/\hat{w}, [])$, so by the induction hypothesis, there exist M_k , w_k , $s_k \neq []$, M' , w' such that $(\hat{M}, 1, \hat{s}) \Rightarrow (M_k, w_k, s_k) \rightarrow (M', w', [])$ and $(M', 1, t) \Rightarrow (V, w'', [])$ for some $w'' > 0$.
By Lemma 59, $(M, 1, s) \rightarrow (\hat{M}, \hat{w}, \hat{s})$, so by Lemma 36 we have $(M, 1, s) \rightarrow (\hat{M}, \hat{w}, \hat{s}) \Rightarrow (M_k, w_k \hat{w}, s_k) \rightarrow (M', w' \hat{w}, [])$ as required.

In either case, the uniqueness follows by Lemma 175 in Appendix E. ■

Lemma 61 . If $(M, w, s) \Rightarrow (M', w', s')$ and $w > 0$ and $M' \neq E[\text{fail}]$, then $w' > 0$

Proof: By induction on the derivation of $(M, w, s) \Rightarrow (M', w', s')$, with appeal to Lemma 28. ■

Restatement of Lemma 58 $\mathbf{P}_{\text{peval}(M, s)}^{\mathcal{V}}(t) = 0$ if and only if $\mathbf{P}_M^{\mathcal{V}}(s@t) = 0$.

Proof: The result follows immediately if $s = []$ (because $\text{peval}(M, []) = M$), so let us assume that $s \neq []$.

- \Rightarrow : For contradiction, let us suppose that $\mathbf{P}_{\text{peval}(M, s)}^{\mathcal{V}}(t) = 0$ and $\mathbf{P}_M^{\mathcal{V}}(s@t) > 0$. Then we have $(M, 1, s@t) \Rightarrow (V, w, [])$ for some V , $w > 0$. By Lemma 60,

$(M, 1, s) \Rightarrow (M_k, w_k, s_k) \rightarrow (M', w', \square)$ for some $M_k, w_k, s_k \neq \square, M', w'$ and $(M', 1, t) \Rightarrow (V', w'', \square)$ for some $V', w'' > 0$.

Hence, $\text{peval}(M, s) = M'$ and $\mathbf{P}_{M'}^{\mathcal{V}}(t) = w'' > 0$, which contradicts the assumption.

- \Leftarrow : Suppose $\mathbf{P}_M^{\mathcal{V}}(s@t) = 0$ and $\mathbf{P}_{\text{peval}(M, s)}^{\mathcal{V}}(t) > 0$. Then, we know that $\text{peval}(M, s) \neq \text{fail}$ (as otherwise we would have $\mathbf{P}_{\text{peval}(M, s)}^{\mathcal{V}}(t) = 0$), so by definition of peval , we have $M_k, w_k, s_k \neq \square, M', w'$ such that $(M, 1, s) \Rightarrow (M_k, w_k, s_k) \rightarrow (M', w', \square)$. Moreover, $(M', 1, t) \Rightarrow (V, w'', \square)$ for some $V \in \mathcal{V}$ and $w'' > 0$. We know that $M' \neq E[\text{fail}]$, because otherwise it would not reduce to a value, so by Lemma 61, $w' > 0$. Meanwhile, Lemma 38 yields $(M, 1, s@t) \Rightarrow (V, w'w'', \square)$, so $\mathbf{P}_M^{\mathcal{V}}(s@t) = w'w'' > 0$, which contradicts the assumption. ■

Now, we show that the density $\mathbf{P}_M^{\mathcal{V}}(s)$ is always smaller or equal to the `score-less` density $\mathbf{P}_M^*(s)$. To formally prove this intuitive property, we need the following lemma:

Lemma 62 *If $(M, 1, s) \Rightarrow (V, w, \square)$ then $(M, 1, s) \rightsquigarrow^* (V, w', \square)$ for some w' . Moreover, $w' \geq w$.*

Proof: By induction on the derivation of $(M, 1, s) \Rightarrow (V, w, \square)$.

- Base case: If $(M, 1, s) = (V, w, \square)$, the result follows immediately.
- Induction step: If $(M, 1, s) \Rightarrow (V, w, \square)$ was derived in more than one step, we have $(M, 1, s) \rightarrow (M', w', s') \rightarrow^k (V, w, \square)$ for some $M, w', s', k \geq 0$. If $(M, 1, s) \rightarrow (M', w', s')$ was derived with (RED RANDOM FAIL), then $w' = 0$ and $M' = E[\text{fail}]$, which is a contradiction, as $E[\text{fail}]$ cannot reduce to a value. Hence $(M, 1, s) \rightarrow (M', w', s')$ was not derived with (RED RANDOM FAIL), so $w' > 0$ by Lemma 28. By Lemma 36, we get $(M', 1, s') \rightarrow^k (V, w/w', \square)$. By induction hypothesis, $(M', 1, s') \rightsquigarrow^* (V, w'', \square)$, where $w'' \geq w/w'$.
 - If $(M, 1, s) \rightarrow (M', w', s')$ was derived with (RED SCORE), then $(M, 1, s) \rightsquigarrow (M', 1, s')$, so $(M, 1, s) \rightsquigarrow (M', 1, s') \rightsquigarrow^* (V, w'', \square)$. Since $w' < 1$ (scores cannot be greater than 1), we have $w'' \geq w/w' \geq w$, as required.
 - Otherwise, we have $(M, 1, s) \rightsquigarrow (M', w', s')$. By an equivalent of Lemma 36 for \rightsquigarrow^* , we have $(M', w', s') \rightsquigarrow^* (V, w'w'', \square)$, so $(M, 1, s) \rightsquigarrow (M', w', s') \rightsquigarrow^* (V, w'w'', \square)$, where $w'w'' \geq w$, as required.

■

We can now prove the property described above.

Lemma 63 For all $s \in \mathbb{U}$, $\mathbf{P}_M^{\mathcal{V}}(s) \leq \mathbf{P}_M^*(s)$

Proof: If there is no $V \in \mathcal{V}$ and $w > 0$ such that $(M, 1, s) \Rightarrow (V, w, \square)$, then $\mathbf{P}_M^{\mathcal{V}}(s) = 0$, so the inequality holds trivially.

If $(M, 1, s) \Rightarrow (V, w, \square)$ for some $V, w > 0$, then by Lemma 62, we have $(M, 1, s) \rightsquigarrow^* (V, w', \square)$ for some $w' \geq w$. Hence, $\mathbf{P}_M^*(s) = w' \geq w = \mathbf{P}_M^{\mathcal{V}}(s)$, as required. ■

We also prove a partial converse of the above property, which states that if $\mathbf{P}_M^{\mathcal{V}}(t) = 0$, then $\mathbf{P}_M^*(t) = 0$.

Lemma 64 If $(M, 1, s) \rightsquigarrow^* (V, w, \square)$ and $w > 0$, then $(M, 1, s) \Rightarrow (V, w', \square)$ for some $w' > 0$.

Proof: Similar to proof of Lemma 62. ■

Lemma 65 If $\mathbf{P}_M^{\mathcal{V}}(t) = 0$, then $\mathbf{P}_M^*(t) = 0$

Proof: Suppose for contradiction that $\mathbf{P}_M^{\mathcal{V}}(t) = 0$ and $\mathbf{P}_M^*(t) > 0$. Then $(M, 1, s) \rightsquigarrow^* (V, w, \square)$ for some $w > 0$. But by Lemma 64, this implies that $(M, 1, s) \Rightarrow (V, w', \square)$ for some $w' > 0$. Hence, $\mathbf{P}_M^{\mathcal{V}}(t) > 0$, which contradicts the assumption. ■

Finally, we show that if the density $q(s, t)$ is zero, then the density of the given program M at t is also zero, and that the converse is also true for $t \neq \square$.

Lemma 66 For all $s, t \in \mathbb{U}$, if $q(s, t) = 0$, then $\mathbf{P}_M^{\mathcal{V}}(t) = 0$

Proof: If $t = \square$, then $\mathbf{P}_M^{\mathcal{V}}(t) = 0$ by assumption, so now let us assume $t \neq \square$.

Let $k = \min(|s|, |t|)$ and $N = \text{peval}(M, s_{1..k})$. We have $q(s, t) = \prod_{i=1}^k \text{pdf}_{\text{Gaussian}}(s_i, \sigma^2, t_i) \mathbf{P}_N^*(t_{k+1..|t|})$. Since the Gaussian pdf is positive everywhere, $q(s, t) = 0$ implies $\mathbf{P}_N^*(t_{k+1..|t|}) = 0$, which by Lemma 63 gives $\mathbf{P}_N^{\mathcal{V}}(t_{k+1..|t|}) = 0$. Hence, $\mathbf{P}_M^{\mathcal{V}}(t) = 0$ by Lemma 58. ■

Lemma 67 For any $s \in \mathbb{U}$, if $t \neq \square$ and $\mathbf{P}_M^{\mathcal{V}}(t) = 0$, then $q(s, t) = 0$.

Proof: Let $k = \min(|s|, |t|)$ and $N = \text{peval}(M, t_{1..k})$. We have $q(s, t) = \prod_{i=1}^k \text{pdf}_{\text{Gaussian}}(s_i, \sigma^2, t_i) \mathbf{P}_N^*(t_{k+1..|t|})$. By Lemma 58, $\mathbf{P}_N^{\mathcal{Y}}(t_{k+1..|t|}) = 0$, so by Lemma 65, $\mathbf{P}_N^*(t_{k+1..|t|}) = 0$, which implies $q(s, t) = 0$. ■

7.3.2 π -Irreducibility

We first prove strong irreducibility, which implies π -irreducibility.

Lemma 68 (Strong Irreducibility) *If $\mathbf{P}_M^{\mathcal{Y}}(s) \neq 0$, then for any B such that $\langle\langle M \rangle\rangle^{\mathcal{Y}}(B) > 0$ we have $P(s, B) > 0$.*

Proof:

For contradiction, let us suppose that $P(s, B) = 0$. We have:

- If $\mathbf{P}_M^{\mathcal{Y}}(s) = 0$, then $\alpha(s, t) = 1$ for all $t \in \mathbb{U}$, so we have $P(s, B) \geq \int_B q(s, t) dt$. Thus, if $P(s, B) = 0$, then $q(s, t) = 0$ for μ -almost-every $t \in \mathbb{U}$, so by Lemma 66, we have $\mathbf{P}_M^{\mathcal{Y}}(t) = 0$ μ -almost everywhere on B , which implies $\int_B \mathbf{P}_M^{\mathcal{Y}}(t) dt = 0$. This contradicts the assumption $\langle\langle M \rangle\rangle^{\mathcal{Y}}(B) > 0$.
- If $\mathbf{P}_M^{\mathcal{Y}}(s) > 0$, then

$$\begin{aligned} P(s, B) &\geq \int_B \alpha(s, t) q(s, t) dt \\ &= \int_B \min \left(\left\{ q(s, t), \frac{\mathbf{P}_M^{\mathcal{Y}}(t) q(t, s)}{\mathbf{P}_M^{\mathcal{Y}}(s)} \right\} \right) dt \\ &= \int_{B_1} q(s, t) dt + \frac{1}{\mathbf{P}_M^{\mathcal{Y}}(s)} \int_{B_2} \mathbf{P}_M^{\mathcal{Y}}(t) q(t, s) dt \end{aligned}$$

where $B_1 = \left\{ t \in B \mid q(s, t) \leq \frac{\mathbf{P}_M^{\mathcal{Y}}(t) q(t, s)}{\mathbf{P}_M^{\mathcal{Y}}(s)} \right\}$ and $B_2 = B \setminus B_1$ (both sets are obviously measurable). If $P(s, B) = 0$, then $\int_{B_1} q(s, t) dt = 0$ and $\int_{B_2} \mathbf{P}_M^{\mathcal{Y}}(t) q(t, s) dt = 0$.

- If $\int_{B_1} q(s, t) dt = 0$, then $q(s, t) = 0$ μ -almost everywhere on B_1 . By Lemma 66, this implies that $\mathbf{P}_M^{\mathcal{Y}}(t) = 0$ μ -almost everywhere on B_1 , so $\int_{B_1} \mathbf{P}_M^{\mathcal{Y}}(t) dt = 0$.
- Similarly, if $\int_{B_2} \mathbf{P}_M^{\mathcal{Y}}(t) q(t, s) dt = 0$, then $\mathbf{P}_M^{\mathcal{Y}}(t) q(t, s) = 0$ μ -almost everywhere on B_2 . By Lemma 66, if $q(t, s) = 0$, then $\mathbf{P}_M^{\mathcal{Y}}(s) = 0$, which contradicts the assumption, so $q(t, s) > 0$ for all $t \in \mathbb{U}$. Hence, $\mathbf{P}_M^{\mathcal{Y}}(t) = 0$ μ -almost everywhere on B_2 , so $\int_{B_2} \mathbf{P}_M^{\mathcal{Y}}(t) dt = 0$.

Thus, we get $\langle\langle M \rangle\rangle^{\mathcal{Y}}(B) = \int_B \mathbf{P}_M^{\mathcal{Y}}(t) dt = 0$, which contradicts the assumption. ■

Corollary 2 (Irreducibility) *The Markov chain P is π -irreducible.*

7.3.3 Aperiodicity

We now prove the aperiodicity of the transition kernel.

Lemma 69 (Aperiodicity) *There do not exist integer $d \geq 2$ and non-empty disjoint sets $B_1, \dots, B_d \in \mathcal{S}$ such that $\pi(B_1) > 0$ and if $s \in B_i$, then $P(s, B_j) = 1$, where $j = i + 1 \bmod d$.*

Proof: For contradiction, suppose the Markov chain is periodic. Let $s \in B_1$. Then $P(s, B_2) = 1$. Since P is a probability kernel, we have $P(s, B_1) \leq P(s, \mathbb{U} \setminus B_2) = 0$. But $\pi(B_1) > 0$ by assumption, so this contradicts strong irreducibility. Hence, the Markov chain P is aperiodic. ■

We have now proven π -irreducibility, which guarantees that the algorithm converges for π -almost-every starting trace s .

7.3.4 Harris recurrence

In order to show that the closure of the transition kernel converges to π for every starting trace s (and not just π -almost-every trace s), we need to prove Harris recurrence.

Lemma 70 (Harris recurrence) *For every $s \in \mathbb{U}$ and every set $B \in \mathcal{S}$, such that $\pi(B) > 0$, the probability that the Markov chain P will reach B in finitely many steps starting from s is 1.*

We will prove this property by using Lemma 57. As usual, we need some additional lemmas. We begin by stating that the probability of a transition from $s \neq []$ to a null set for π not containing s and the empty trace is always 0.

Lemma 71 *If $\pi(B) = 0$ and $s \neq []$, then for any $s \neq []$, $P(s, B \setminus \{s, []\}) = 0$.*

Proof: Let $B' = B \setminus \{s, \square\}$. As $\pi(B) = 0$ implies $\pi(B') = 0$, we have $\int_{B'} \mathbf{P}_M^\gamma(t) dt = 0$, and so $\mathbf{P}_M^\gamma(t) = 0$ μ -almost-everywhere on B' . Hence, by Lemma 67, for every s , $q(s, t) = 0$ for μ -almost-every $t \in B'$. Thus, we have:

$$\begin{aligned} P(s, B') &= \int_{B'} \alpha(s, t) q(s, t) dt \\ &\leq \int_{B'} q(s, t) dt \\ &= 0 \end{aligned}$$

■

Corollary 3 *If $\pi(B) = 0$, then for any $s \in \mathbb{U}$, $P(s, B) \leq P(s, \{s, \square\})$*

Similarly, we prove that the transition from \square to a null set for π is not possible, unless that null set contains \square (in which case proposing an invalid trace will result in the algorithm staying in \square). Note that we assume that in the algorithm, the initial trace has positive density, and so is not empty, so this and several subsequent lemmas are only proven for mathematical completeness.

Lemma 72 *If $\pi(B) = 0$, then $P(\square, B \setminus \{\square\}) = 0$.*

Proof: Let $B' = B \setminus \{\square\}$. Since $\pi(B) = 0$ implies $\pi(B') = 0$, we have $\int_{B'} \mathbf{P}_M^\gamma(t) dt = 0$, and so $\mathbf{P}_M^\gamma(t) = 0$ μ -almost-everywhere on B' . By Lemma 65, $\mathbf{P}_M^*(t) = 0$ for μ -almost-every $t \in B'$. Hence.

$$\begin{aligned} P(\square, B') &= \int_{B'} \alpha(\square, t) q(\square, t) dt \\ &\leq \int_{B'} q(\square, t) dt \\ &= \int_{B'} \mathbf{P}_M^*(t) dt \\ &= 0 \end{aligned}$$

■

Corollary 4 *If $\pi(B) = 0$, then $P(\square, B) \leq P(\square, \{\square\})$.*

Now, we show that the Markov chain P cannot stay at the empty trace indefinitely.

Lemma 73 *$P(\square, \{\square\}) < 1$.*

Proof: We have $P(\square, \{\square\}) = \alpha(\square, \square)q(\square, \square)\mu(\{\square\}) + \int (1 - \alpha(\square, t))q(\square, t) dt = q(\square, \square)$, as $\mu(\{\square\}) = 1$ and $\alpha(\square, t) = 1$ for every t . Hence, $P(\square, \{\square\}) = 1 - \int_{\mathbb{U} \setminus \{\square\}} q(\square, t) dt$. Since we assume that $\mathbf{P}_M^\gamma(\square) = 0$ and $\int \mathbf{P}_M^\gamma(s) ds > 0$, by Lemma 66 we have $\int_{\mathbb{U} \setminus \{\square\}} q(\square, t) dt > 0$, and so $P(\square, \{\square\}) < 1$, as required. ■

Similarly, we show that the probability of moving out of the current non-empty trace is positive.

Lemma 74 *If $s \neq \square$, then $P(s, \{s\}) < 1$.*

Proof: We have $P(s, \{s\}) = \alpha(s, s)q(s, s)\mu(\{s\}) + \int (1 - \alpha(s, t))q(s, t) dt$. Since the measure μ is zero on any singleton set other than $\{\square\}$, this simplifies to $P(s, \{s\}) = \int (1 - \alpha(s, t))q(s, t) dt$.

If $\mathbf{P}_M^\gamma(s) = 0$, then $\alpha(s, t) = 1$ for every $t \in \mathbb{U}$, so $P(s, \{s\}) = 0$.

If $\mathbf{P}_M^\gamma(s) > 0$ then we have $P(s, \{s\}) = 1 - \int \alpha(s, t)q(s, t) dt$, so $P(s, \{s\}) < 1$ by the same reasoning as in the proof of Lemma 68 (taking $B = \mathbb{U}$ and using the assumption that $\int \mathbf{P}_M^\gamma(t) dt > 0$). ■

We also show that the algorithm cannot move to the empty trace from a non-empty trace with probability 1. Note that such a transition is only ever possible if the starting state s is invalid.

Lemma 75 *If $s \neq \square$, then $P(s, \{\square\}) \leq 1$.*

Proof: We have $P(s, \{\square\}) = \alpha(s, \square)q(s, \square)$. If $\mathbf{P}_M^\gamma(s) > 0$, then $\alpha(s, \square) = 0$, so $P(s, \{\square\}) = 0$. If $\mathbf{P}_M^\gamma(s) = 0$, then $P(s, \{\square\}) = q(s, \square) = 1 - \int_{\mathbb{U} \setminus \{\square\}} q(s, t) dt < 1$, by the assumptions that $\int \mathbf{P}_M^\gamma(t) dt > 0$ and $\mathbf{P}_M^\gamma(\square) = 0$ and Lemma 66. ■

Recall that $\underline{P}^n(s, B)$ is the probability of all n first elements of the Markov chain P , as well as the starting element s , being in the set B .

We can now prove that the probability of staying in a set of zero probability goes to 0 as the number of steps goes to infinity. We first show the following useful lemma:

Lemma 76 *If $\pi(B) = 0$, then for every $n \in \mathbb{N}$, $\underline{P}^n(\square, B^n) \leq (P(\square, \{\square\}))^n$.*

Proof: By induction on n .

- Base case: $n = 1$: We have $\underline{P}^1(\square, B) = P(\square, B) \leq P(\square, \{\square\})$ by Corollary 4.

- Induction step: We have $\underline{P}^{n+1}(\square, B^{n+1}) = \int_B \underline{P}^n(t, B^n) P(\square, dt) \leq \int_{\{\square\}} \underline{P}^n(t, B^n) P(\square, dt) = \underline{P}^n(\square, B^n) P(\square, \square) \leq P(\square, \square)^{n+1}$ by the induction hypothesis.

■

We can now show the above property.

Lemma 77 *If $\pi(B) = 0$, then for every $s \in \mathbb{U}$, there exists $c_s \in [0, 1)$ such that for every $n \in \mathbb{N}$, $\underline{P}^n(s, B^n) \leq c_s^n$.*

Proof: Fix $s \in B$. If $s = \square$, the result follows immediately by Lemma 76 and Lemma 73. Now, assume that $s \neq \square$. Define c_s to be:

$$c_s = \max(\{P(s, \{s\}), P(s, \{\square\}), P(\square, \{\square\})\})$$

By Lemmas 73, 74 and 75, $c_s < 1$.

We can now prove the statement by induction on n .

- Base case: $n = 1$: We have $\underline{P}^1(s, B) = P(s, B) \leq P(s, \{s, \square\}) = P(s, \{s\}) + P(s, \{\square\})$. If $\mathbf{P}_M^\gamma(s) = 0$, then $P(s, \{s\}) = \int (1 - \alpha(s, t)) q(s, t) dt = 0$ (because $\alpha(s, t) = 1$ for all t), and if $\mathbf{P}_M^\gamma(s) > 0$, then $P(s, \{\square\}) = \alpha(s, \square) q(s, \square) = 0$ (because if $q(s, \square) > 0$, then $\alpha(s, \square) = 0$). In either case, $\underline{P}^1(s, B) \leq c_s$.
- Induction step: We have $\underline{P}^{n+1}(s, B^{n+1}) = \int_B \underline{P}^n(t, B^n) P(s, dt) \leq \int_{\{s, \square\}} \underline{P}^n(t, B^n) P(s, dt)$ by absolute continuity. Hence, $\underline{P}^{n+1}(s, B^{n+1}) \leq \underline{P}^n(s, B^n) P(s, \{s\}) + \underline{P}^n(\square, B^n) P(s, \{\square\})$.
 - If $\mathbf{P}_M^\gamma(s) = 0$, then $P(s, \{s\}) = 0$, so $\underline{P}^{n+1}(s, B^{n+1}) \leq \underline{P}^n(\square, B^n) P(s, \{\square\})$. By Lemma 76, $\underline{P}^n(\square, B^n) \leq c_s^n$. Hence, $\underline{P}^{n+1}(s, B^{n+1}) \leq c_s^{n+1}$, as required.
 - If $\mathbf{P}_M^\gamma(s) > 0$, then $P(s, \{\square\}) = 0$, so $\underline{P}^{n+1}(s, B^{n+1}) \leq \underline{P}^n(s, B^n) P(s, \{s\}) = \underline{P}^n(s, B^n) P(s, \{s\})$. Hence, we get $\underline{P}^{n+1}(s, B^{n+1}) \leq c_s^{n+1}$ immediately by the induction hypothesis.

■

From the above results and Lemma 57, we get Harris recurrence.

Lemma 78 *The Markov chain P is Harris recurrent.*

Proof: By Lemma 77, for every $s \in \mathbb{U}$ and $B \in \mathcal{S}$ such that $\pi(B) = 0$, there exists $c_s \in [0, 1)$ such that for each $n \in \mathbb{N}$, $\underline{P}^n(s, B^n) \leq c_s^n$, so $\lim_{n \rightarrow \infty} \underline{P}^n(s, B^n) = 0$ (that is, the probability of the chain making a transition to a trace in B in every step is 0). Hence, by Lemma 57, the chain P is Harris recurrent. ■

We can now prove the main theorem of this chapter. We begin by using Lemma 56 and the results shown in this section to prove convergence of the kernel P on traces.

Lemma 79 *For every $s \in \mathbb{U}$, $\lim_{n \rightarrow \infty} \|P^n(x, \cdot) - \pi\| = 0$*

Proof: By Lemma 55, π is the stationary distribution of P . By Corollary 2, P is π -irreducible, by Lemma 69, P is aperiodic and by Lemma 78, it is also Harris recurrent. Hence, by Lemma 56, we have $\lim_{n \rightarrow \infty} \|P^n(x, \cdot) - \pi\| = 0$ for every $s \in \mathbb{U}$. ■

We use the following useful property of the variational norm to obtain a convergence result for output values.

Lemma 80 *If (X_1, Σ_1) and (X_2, Σ_2) are measurable spaces and μ_1 and μ_2 are probability measures on (X_1, Σ_1) and $f : X_1 \rightarrow X_2$ is measurable Σ_1/Σ_2 and satisfies $f^{-1}(X_2) = X_1$, then*

$$\|\mu_1 f^{-1} - \mu_2 f^{-1}\| \leq \|\mu_1 - \mu_2\|$$

Proof: We have $\sup_{B \in \Sigma_2} |\mu_1 f^{-1}(B) - \mu_2 f^{-1}(B)| = \sup_{A \in \Sigma'_1} |\mu_1(A) - \mu_2(A)|$, where $\Sigma'_1 = \{f^{-1}(B) | B \in \Sigma_2\}$. By measurability of f we get $\Sigma'_1 \subseteq \Sigma_1$, so by monotonicity of sup we get $\sup_{A \in \Sigma'_1} |\mu_1(A) - \mu_2(A)| \leq \sup_{A \in \Sigma_1} |\mu_1(A) - \mu_2(A)|$. ■

From the above two lemmas, we get correctness of the main theorem.

Restatement of Theorem 6 *For every $s \in \mathbb{U}$,*

$$\lim_{n \rightarrow \infty} \|T^n(s, \cdot) - \widehat{\llbracket M \rrbracket_{\mathbb{U}}|_{\mathcal{V}}}\| = 0$$

Proof: By Lemma 79, we have $\lim_{n \rightarrow \infty} \|P^n(x, \cdot) - \pi\| = 0$.

$$\begin{aligned} & \text{By definition, } T^n(s, A) = P^n(s, \mathbf{O}_M^{-1}(A)) \text{ and } \widehat{\llbracket M \rrbracket_{\mathbb{U}}|_{\mathcal{V}}}(A) = \frac{\llbracket M \rrbracket_{\mathbb{U}}|_{\mathcal{V}}(A)}{\llbracket M \rrbracket_{\mathbb{U}}|_{\mathcal{V}}(\mathcal{G}\mathcal{V})} \\ &= \frac{\langle\langle M \rangle\rangle^{\mathcal{V}}(\mathbf{O}_M^{-1}(A))}{\langle\langle M \rangle\rangle^{\mathcal{V}}(\mathbf{O}_M^{-1}(\mathcal{G}\mathcal{V}))} = \frac{\langle\langle M \rangle\rangle^{\mathcal{V}}(\mathbf{O}_M^{-1}(A))}{\langle\langle M \rangle\rangle^{\mathcal{V}}(\mathbb{U})} = \pi(\mathbf{O}_M^{-1}(A)). \end{aligned}$$

Thus, by Lemma 80 and the squeeze theorem for limits we get

$$\lim_{n \rightarrow \infty} \|T^n(s, \cdot) - \widehat{\llbracket M \rrbracket_{\mathbb{U}}|_{\mathcal{V}}}\| \leq \lim_{n \rightarrow \infty} \|P^n(s, \cdot) - \pi\| = 0.$$

■

7.4 Examples

In this section, we illustrate the behaviour of the inference algorithm in the presence of hard and soft conditioning with several examples. Specifically, we revisit the geometric distribution example from Section 6.1.3 and the two implementations of linear regression from Section 6.1.4.

7.4.1 Geometric Distribution

We begin with the program implementing the geometric distribution (which we will call M_{geom} from now on):

```

let geometric =
  (fix g.
    λp. (let z = rnd() in
      let y = (z < p) in
      if y then 0 else 1 + (g p))) in
let n = geometric 0.5 in
let b = n > 1 in
if b then n else fail

```

This example does not use soft conditioning, and the only distribution sampled from is *rnd*, whose density is equal to 1 on the whole unit interval, so the density $\mathbf{P}_{M_{geom}}^{\mathcal{V}}(s)$ must be 1 for every valid trace s and 0 for every invalid trace. It is easy to see that a non-empty trace with all elements in the unit interval is valid if and only if its last element is greater or equal 0.5 (setting *y* to *false*), all other elements are less than 0.5 (setting *y* to *true*) and its length is greater than 2 (because of the conditioning at the end). Thus, the set of valid traces is precisely $S_{geom} = \{s \mid s_i \in [0, 0.5) \text{ for } i < |s| \wedge s_{|s|} \in [0.5, 1] \wedge |s| > 2\}$. We have $\mathbf{P}_{M_{geom}}^{\mathcal{V}}(s) = [s \in S_{geom}]$. Moreover, because *score* is not used in this model, $\mathbf{P}_{M_{geom}}^*(s) = \mathbf{P}_{M_{geom}}^{\mathcal{V}}(s) = [s \in S_{geom}]$.

The transition density is $q(s, t) = (\prod_{i=1}^k \text{pdf}_{\text{Gaussian}}(s_i, \sigma^2, t_i)) \mathbf{P}_{M'}^*(t_{k+1..|t|})$, where $k = \min\{|s|, |t|\}$ and $M' = \text{peval}(M_{geom}, s_{1..k})$. By Lemma 58, $\mathbf{P}_{M'}^{\mathcal{V}}(t_{k+1..|t|}) = 0$ if and only if $\mathbf{P}_{M_{geom}}^{\mathcal{V}}(t) = 0$, which, in the absence of soft conditioning and draws from distributions with non-constant densities, means $\mathbf{P}_{M'}^*(t_{k+1..|s|}) = \mathbf{P}_{M'}^{\mathcal{V}}(t_{k+1..|s|}) = \mathbf{P}_{M_{geom}}^{\mathcal{V}}(t)$. Hence, the density simplifies to $q(s, t) = (\prod_{i=1}^k \text{pdf}_{\text{Gaussian}}(s_i, \sigma^2, t_i)) [t \in S_{geom}]$.

If $\mathbf{P}_{M_{geom}}^{\mathcal{V}}(s)q(s,t) = 0$, then $\alpha(s,t) = 1$. Otherwise, if $t \neq []$, we have:

$$\alpha(s,t) = \min \left\{ 1, \frac{[t \in S_{geom}](\prod_{i=1}^k \text{pdf}_{\text{Gaussian}}(t_i, \sigma^2, s_i))[s \in S_{geom}]}{[s \in S_{geom}](\prod_{i=1}^k \text{pdf}_{\text{Gaussian}}(s_i, \sigma^2, t_i))[t \in S_{geom}]} \right\} = 1$$

because the products of Gaussian densities obviously cancel out and $\mathbf{P}_{M_{geom}}^{\mathcal{V}}(s)q(s,t) > 0$ implies that $s \in S_{geom}$ and $t \in S_{geom}$.

We can easily check that $q(s, []) = 1 - \int_{\mathbb{W} \setminus \{[]\}} q(s,t) dt > 0$ (note that the probability of reaching fail and proposing an empty trace is positive because of the hard conditioning), so if $\mathbf{P}_{M_{geom}}^{\mathcal{V}}(s) > 0$, then $\alpha(s, []) = \min \left\{ 1, \frac{\mathbf{P}_{M_{geom}}^{\mathcal{V}}([])q([],s)}{\mathbf{P}_{M_{geom}}^{\mathcal{V}}(s)q(s,[])} \right\} = 0$. Thus, the acceptance ratio is:

$$\alpha(s,t) = \begin{cases} 0 & \text{if } [t = []] \text{ and } [s \in S_{geom}] \\ 1 & \text{otherwise} \end{cases}$$

This effectively means that every valid trace is accepted. The proposal kernel of the algorithm for this model is

$$Q(s,B) = \int_{B \cap S_{geom}} (\prod_{i=1}^{\min\{|s|,|t|\}} \text{pdf}_{\text{Gaussian}}(s_i, \sigma^2, t_i)) dt$$

for $[] \notin B$. We know that $1 - \alpha(s,t) = 1$ if and only if $t = []$ and $s \in S_{geom}$, and $1 - \alpha(s,t) = 0$ otherwise, so the transition kernel is:

$$\begin{aligned} P(s,B) &= \int_{B \cap S_{geom}} \prod_{i=1}^{\min\{|s|,|t|\}} \text{pdf}_{\text{Gaussian}}(s_i, \sigma^2, t_i) dt \\ &\quad + [s \in B] \left(\int [s \in S_{geom}] [t = []] q(s,t) dt \right) \\ &= \int_{B \cap S_{geom}} \prod_{i=1}^{\min\{|s|,|t|\}} \text{pdf}_{\text{Gaussian}}(s_i, \sigma^2, t_i) dt \\ &\quad + [s \in B \cap S_{geom}] q(s, []) \\ &= \int_{B \cap S_{geom}} \prod_{i=1}^{\min\{|s|,|t|\}} \text{pdf}_{\text{Gaussian}}(s_i, \sigma^2, t_i) dt \\ &\quad + [s \in B] \left(1 - \int_{S_{geom}} (\prod_{i=1}^{\min\{|s|,|t|\}} \text{pdf}_{\text{Gaussian}}(s_i, \sigma^2, t_i)) dt \right) \end{aligned}$$

for $[] \notin B$.

7.4.2 Linear Regression with flip

Let us now consider the version of the linear regression model from section 6.1.4 which uses `flip` instead of `soft conditioning`. This model takes the following form when translated to the core calculus (assuming we have a function and taking an arbitrary number of arguments):

```

let  $sqr = \lambda x. x * x$  in
let  $squash = \lambda x. \lambda y. \exp(-(sqr(x - y)))$  in
let  $flip = \lambda p. \text{rnd}() < p$  in
let  $softeq = \lambda x. \lambda y. flip(squash\ x\ y)$  in
let  $m = \text{Gaussian}(0, 2)$  in
let  $b = \text{Gaussian}(0, 2)$  in
let  $f = \lambda x. m * x + b$  in
let  $cond = \text{and}((softeq\ (f\ 0)\ 0), (softeq\ (f\ 1)\ 1), (softeq\ (f\ 2)\ 4), (softeq\ (f\ 3)\ 6))$  in
if  $cond$  then  $(f\ 4)$  else fail

```

Every valid trace in this model (which we will call M_{flip}) consists of two values drawn from $\text{Gaussian}(0, 2)$, which are assigned to variables m and b , followed by four values drawn from $\text{rnd}()$ while evaluating the four calls to $softeq$. The hard constraint is satisfied if and only if all calls to $softeq$ return true. The expression $flip(squash\ x\ y)$ evaluates to true if the value sampled from rnd is in the interval $[0, e^{-(x-y)^2})$. Because the density of $\text{rnd}()$ is constant on the unit interval, on any trace of length 6 the density $\mathbf{P}_{M_{flip}}^{\mathcal{V}}(s)$ of the program depends only on the first two elements of the trace s (assumed to be drawn from Gaussians) and on whether the remaining four elements are in the “correct” intervals. Obviously, the density $\mathbf{P}_{M_{flip}}^{\mathcal{V}}$ is zero on traces of other lengths.

The full density of the above program is:

$$\mathbf{P}_{M_{flip}}^{\mathcal{V}}(s) = \begin{cases} (\prod_{i=1}^2 \text{pdf}_{\text{Gaussian}}(0, 2, s_i)) \cdot \left(\prod_{i=1}^4 \left[s_{i+2} \in [0, e^{-(s_1 \cdot x_i + s_2 - y_i)^2}] \right] \right) & \text{if } |s| = 6 \\ 0 & \text{otherwise} \end{cases}$$

where x_i and y_i are the coordinates of the subsequent observed points. Since this example, like the previous one, uses no soft conditioning, $\mathbf{P}_{M_{flip}}^*(s) = \mathbf{P}_{M_{flip}}^{\mathcal{V}}(s)$.

We can now derive the proposal density $q(s, t)$. First, let us suppose that $\mathbf{P}_{M_{flip}}^{\mathcal{V}}(s) > 0$ and $\mathbf{P}_{M_{flip}}^{\mathcal{V}}(t) > 0$ (which implies $|s| = 6$ and $|t| = 6$, as shown above). The formula for $q(s, t)$ then has the following form:

$$q(s, t) = (\prod_{i=1}^6 \text{pdf}_{\text{Gaussian}}(s_i, \sigma^2, t_i)) \mathbf{P}_{M'}^*(\square)$$

where $M' = \text{peval}(M_{flip}, t)$. If $M' \neq \text{fail}$ (which is the case if all of s_3, \dots, s_6 are in the support of rnd) then the program M' is deterministic and has the form:

```

if  $cond$  then  $(s_1 * 4 + s_2)$  else fail

```

where *cond* is true if and only if the condition in the original program was satisfied. Thus, is easy to see that $\mathbf{P}_{M'}^*([]) = 1$ if *cond* = true (which also implies $M' \neq \text{fail}$) and $\mathbf{P}_{M'}^*([]) = 0$ otherwise. Hence, the proposal density expands to:

$$q(s, t) = \left(\prod_{i=1}^6 \text{pdf}_{\text{Gaussian}}(s_i, \sigma^2, t_i) \right) \cdot \left(\prod_{i=1}^4 [t_{i+2} \in [0, e^{-(t_1 \cdot x_i + t_2 - y_i)^2}]] \right)$$

if $s, t \in \mathbb{R}^6$. In case $s \in \mathbb{R}^6$ and $t \notin \mathbb{R}^6$, the density $q(s, t)$ is only non-zero if $t = []$ in which case $q(s, [])$ is the probability of proposing a trace violating the hard constraint. The definition of q is extended to $s \notin \mathbb{R}^6$ for purely technical reasons, as invalid states are unreachable, so we omit the formula for this case.

For $s, t \in \mathbb{R}^6$, the acceptance ratio is:

$$\alpha(s, t) = \min \left\{ 1, \frac{\prod_{i=1}^2 \text{pdf}_{\text{Gaussian}}(0, 2, t_i)}{\prod_{i=1}^2 \text{pdf}_{\text{Gaussian}}(0, 2, s_i)} \right\}$$

if $\mathbf{P}_{M_{\text{flip}}}(s)q(s, t) > 0$ and $\alpha(s, t) = 1$ otherwise (note that the densities $\text{pdf}_{\text{Gaussian}}(s_i, \sigma^2, t_i)$ and $\text{pdf}_{\text{Gaussian}}(t_i, \sigma^2, s_i)$ cancel out and that the assumption $\mathbf{P}_{M_{\text{flip}}}(s)q(s, t) > 0$ implies that the hard condition is satisfied for both s and t). If $t = []$ and s is a valid trace, the acceptance ratio is 0, as expected — because the proposal kernel may propose an invalid trace, $q(s, []) > 0$ in general, so $\mathbf{P}_{M_{\text{flip}}}(s)q(s, []) > 0$, and $\mathbf{P}_{M_{\text{flip}}}([]) = 0$ in the formula for the acceptance ratio. The formulas for the proposal kernel Q and the transition kernel P can be obtained by plugging the definitions of $q(s, t)$ and $\alpha(s, t)$ into the definitions of $Q(s, B)$ and $P(s, B)$, like in the previous example.

Note that a proposed trace t is only valid if all values t_i for $i \geq 2$ are within certain small intervals. As these values are proposed from Gaussian distributions, with unbounded supports, this may be very unlikely to happen, especially if the number of data points is much larger. This shows that simulating soft conditioning with hard constraints, while semantically correct, is very inefficient, and makes the case for adding primitives for soft conditioning to probabilistic languages.

7.4.3 Linear Regression with score

In this updated version of the linear regression model, the *softeq* function is redefined to call *score* instead of rejecting the trace with a given probability. This means that all traces which can be proposed by the algorithm are valid, but still have different weights, affecting the acceptance probability.

The updated program, M_{score} , is the same as the previous one, except for the definition of *softeq*:

```

...
let softeq = λx. λy. score(squash x y) in
...

```

This model does away with the additional random variables used previously to perform conditioning, so every valid trace consists of just two values, both drawn from $\text{Gaussian}(0, 2)$. For any valid trace, the density $\mathbf{P}_{M_{\text{score}}}^{\mathcal{V}}$ is a product of the two Gaussian densities and the values of the four arguments passed to `score`:

$$\mathbf{P}_{M_{\text{score}}}^{\mathcal{V}}(s) = \prod_{i=1}^2 \text{pdf}_{\text{Gaussian}}(0, 2, s_i) \prod_{i=1}^4 e^{-(s_1 \cdot x_i + s_2 - y_i)^2}$$

The density $q(s, t)$ for $s \in \mathbb{R}^2$ and $t \in \mathbb{R}^2$ has the following form:

$$q(s, t) = \prod_{i=1}^2 \text{pdf}_{\text{Gaussian}}(s_i, \sigma^2, t_i) \mathbf{P}_{M'}^*([])$$

where $M' = \text{peval}(M_{\text{score}}, s)$. As the function $\mathbf{P}_{M'}^*$ ignores arguments passed to `score`, and M' does not contain any random draws and returns a value, $\mathbf{P}_{M'}^*([]) = 1$. Hence, the density is:

$$q(s, t) = \prod_{i=1}^2 \text{pdf}_{\text{Gaussian}}(s_i, \sigma^2, t_i)$$

If $t \notin \mathbb{R}^2$, the density is 0 (including the case $t = []$, because $\int_{\mathbb{W} \setminus \{[]\}} q(s, t) dt = 1$).

The acceptance ratio for $s \in \mathbb{R}^2$ and $t \in \mathbb{R}^2$ is:

$$\alpha(s, t) = \min \left\{ 1, \frac{\prod_{i=1}^2 \text{pdf}_{\text{Gaussian}}(0, 2, t_i) \prod_{i=1}^4 e^{-(t_1 \cdot x_i + t_2 - y_i)^2}}{\prod_{i=1}^2 \text{pdf}_{\text{Gaussian}}(0, 2, s_i) \prod_{i=1}^4 e^{-(s_1 \cdot x_i + s_2 - y_i)^2}} \right\}$$

since the proposal density is symmetric and so $q(s, t)$ and $q(t, s)$ cancel out. If $s \in \mathbb{R}^2$ and $t \notin \mathbb{R}^2$, then $\alpha(s, t) = 1$, but the proposal density is always zero for these traces.

This implementation of linear regression is obviously much more efficient than the one using *flip*. Obviously, traces with lower scores are still more likely to be rejected, but every proposed trace is valid and the acceptance probability depends on the ratio of trace scores, rather than raw scores, which means that the algorithm is more likely to move to a region with similar or higher probability.

7.5 Discussion of the Algorithm

In this section we discuss the motivation for using the particular version of Metropolis-Hastings presented here, and also highlight some deficiencies of the algorithm we use.

7.5.1 Motivation for Using Multi-Site Inference

Unlike the algorithm presented by Wingate et al. [2011], and like the one used by Hur et al. [2015], our version of Metropolis-Hastings is *multi-site*, in that it resamples all elements of the trace at each step. This choice was made to ensure that the proposal kernel has a density with respect to the stock measure on traces. The proposal kernel in [Wingate et al., 2011], which only redraws one random variable in a program and possibly variables depending on it, does not have a density, because any reasonable measure on traces (or databases of random values, as they are called in the aforementioned paper) is zero on the set of traces in which the value of at least one continuous variable is fixed. In other words, such a kernel is not absolutely continuous with respect to the stock measure on program traces.

The choice of a proposal kernel with a density simplifies the proofs significantly and allows us to use the standard framework for reasoning about Metropolis-Hastings [Tierney, 1994, Roberts et al., 2004], without having to worry about the reversibility of the constructed Markov chain. Alternatively, we could have used the results from [Tierney, 1998] to construct a density-less proposal kernel yielding a reversible chain, but the lack of density would have also complicated the proofs of irreducibility and Harris recurrence.

Admittedly, a multi-site Metropolis-Hastings algorithm is in general less efficient. In programs with multiple hard constraints involving many variables, such as the linear regression with `flip` from Section 7.4.2, the probability that at least one condition will not be satisfied after resampling all random variables can be very high. Even without hard constraints, the probability that the weight of at least one newly sampled value will be close to 0, and reduce the trace weight, can also be significant. Because of this, the acceptance rate of the multi-site Metropolis-Hastings can be very low. Moreover, in a multi-site algorithm more computations need to be performed to generate a trace, and there is less scope for using optimization techniques such as slicing [Yang et al., 2013, Hur et al., 2014].

7.5.2 Problem With Identifying Random Variables

A significant problem with the algorithm presented here is the way random variables are identified in a trace. For simplicity, we have used linear traces, in which each random variable is identified just by its position in a trace. This means that if the previous trace was s , each i -th element of a new trace t (for $i \leq |s|$) is sampled from

a Gaussian centred at the i -th element of s . However, because programs may have branches, there is no guarantee that the i -th distribution reached when evaluating the program with s will be the same as the i -th distribution encountered while following t . Hence, the value of t_i , proposed from $\text{Gaussian}(s_i, \sigma^2)$, may be a value very unlikely to be drawn from the corresponding distribution in the program, or even outside its support.

For example, consider the following program (which does not represent any useful machine learning model, but is a simple program illustrating problems which may occur in larger, real-world models):

```
let flip =  $\lambda p. \text{rnd}() < p$  in
let x = flip 0.5 in
let y = if x then Gaussian(0, 1) else 5 in
Gaussian(10, 1)
```

Suppose the previous trace was $s = [0.7, 10.2]$ — this is a reasonably likely trace in the program, since the second element is likely to have been drawn from $\text{Gaussian}(10, 1)$, the second distribution reached when *flip* evaluates to false. Now, suppose the new value of the first random variable is $t_1 = 0.3$. Then *flip* 0.5 returns true, so the next distribution reached is $\text{Gaussian}(0, 1)$. The algorithm will sample the new proposed value t_2 for the random draw $\text{Gaussian}(0, 1)$ from $\text{Gaussian}(10.2, \sigma^2)$, the distribution centred around the previous value of $\text{Gaussian}(10, 1)$. Obviously, this value is very unlikely and the resulting trace will have a very low density and will almost certainly be rejected. This means that we will, in practice, never choose the then branch having started with else.

Note that this problem does not invalidate the proof of correctness. Because the Gaussian proposal distribution has infinite support, the probability of choosing the then branch in the example above, while very low, is technically not zero, so strong irreducibility still holds. The main theorem states only asymptotic convergence, so the low probability of transition to another region in the state space is outweighed by the number of samples going to infinity.

While identifying random variables in functional programs is inherently tricky, it is possible to define a more efficient variable naming scheme. One possible solution would be to use a labelled λ -calculus, such as the one proposed (for call-by-name evaluation) by Lévy [1978] and adapted to call-by-value reduction by Blanc [2008]. In such a calculus, each subexpression of a program has a unique label, and reduction

combines the labels of all “active” terms and assigns the resulting label to the reduced term. Hence, the labels in a partially evaluated expression represent the evaluation histories of the subterms. By using such a calculus, we could represent traces as maps from labels to values, rather than lists, and use the concatenation of labels on the path in the expression tree going down to a random draw as an index of this random draw. This way, we would use a value from the previous trace if and only if it was actually drawn from the same distribution.

An attempt at defining a naming scheme for a functional probabilistic language has already been made in the original paper describing lightweight MCMC for probabilistic programs [Wingate et al., 2011], which presents an elegant source-to-source transformation adding a parameter representing the current location to every function. However, this naming system is not powerful enough to properly distinguish random variables in a λ -calculus. Moreover, such a source-to-source transformation would make the proofs more difficult.

At any rate, the design of an efficient variable identification scheme is left as future work.

7.6 Conclusions

In this chapter, we have defined a variant of the Metropolis-Hastings algorithm for the probabilistic λ -calculus presented in Chapter 6 and proven that the distribution of samples generated by the algorithm converges asymptotically to the semantics of the given program. To our best knowledge, it is the first such proof for a functional language with recursion. Because of the use of linear traces, the algorithm is inefficient (although still correct) for certain programs with structural choice, but this could be fixed easily by using a more elaborate variable identification scheme.

Individual Contributions

The proof of correctness of Metropolis-Hastings for the probabilistic lambda-calculus was done in collaboration with other authors of [Borgström et al., 2016], although the idea of using the literature on Metropolis-Hastings on general state spaces was my own. The corrections applied to the proof shown in the paper are entirely my own work. Specifically, I have corrected the proposal density q (the original paper used the naive density $q(s, t) = \prod_{i=1}^k \text{pdf}_{\text{Gaussian}}(s_i, \sigma^2, t_i) \mathbf{P}_N^{\mathcal{Y}}([t_{k+1}, \dots, t_{|t|}])$ where $k = \min\{|s|, |t|\}$, as

described in Section 7.1) and presented a more rigorous proof of irreducibility and Harris recurrence.

Chapter 8

Conclusions

This dissertation aims to advance the state of probabilistic programming by examining the issues of clean design, correctness and trustworthiness of probabilistic languages.

Specifically, we first present a new version of the Tabular schema-based programming language. Its clean meta-theory, rigorously defined semantics and a structural, dependent type system catching common modelling errors make sure models expressible in this language have clearly defined meaning. We prove two theoretical results which show that the language behaves correctly. Moreover, we extend the language with a formula notation for expressing hierarchical linear regressions and define the semantics of the resulting language by translation to pure Tabular.

Secondly, we study the meaning of models represented in a universal, Turing complete functional language, by giving a measure-theoretic semantics to an untyped probabilistic lambda-calculus with continuous distributions and soft and hard conditioning, defining a sub-probability distribution on output values. While our operational approach does not solve the problem of defining distributions on functions in such languages, as values (including functions) are treated purely syntactically, we believe that it is an important step towards understanding higher-order, Turing-complete languages.

Finally, we address the question of trustworthiness and reliability of probabilistic languages by defining a variant of the Metropolis-Hastings algorithm for the aforementioned calculus and formally proving its correctness, ensuring that the distribution of samples converges to the distribution defined by the semantics program. We believe that verification of inference algorithms for probabilistic programs is important, especially in an era where machine learning, and probabilistic programming, are increasingly being used in safety-critical applications. Unlike testing an algorithm on a limited example suite, presenting a formal proof thereof ensures that it cannot fail in

some corner cases, which may come up unexpectedly when using the given system to solve a real-life problem.

While this dissertation has considered two separate aspects of probabilistic programming, design of real-life probabilistic languages and semantics of a higher-order foundational calculus, they are in fact interdependent: good, clean language design should be driven by language semantics and, where applicable, type systems. Conversely, the semantics of calculi and their desirable theoretical properties should be inspired by the applications of these calculi, as foundations of real-world programming languages. Clean probabilistic language design, semantics of universal probabilistic languages and the proofs of correctness of inference algorithms are all stepping stones to the goal of making probabilistic programming a trusted tool for Bayesian inference.

Further Work Many unsolved problems still remain in the area of probabilistic programming. Here we describe how the work presented in this dissertation, and related developments in probabilistic programming, can be extended and improved upon.

The Tabular Language Since its inception, the Tabular language has been extended with hierarchical generalised linear models, yielding Fabular [Borgström et al., 2015]. Several other extensions are also possible. One idea would be to add direct support for inference in time series, which are difficult to express in the present version, because of the assumption that random variables defined by all rows are identically independently distributed.

Another possibility is extending the lattice of binding times with indices to allow multiple runs of inference in Tabular models—this way, the user could write complex, nested models, in which a `qry` variable from the i -th run of inference could be treated as a deterministic variable in the $i + 1$ -th run.

A very interesting development would be implementing automatic model suggestion in Tabular. This could be done by following the approach of Nori et al. [2015], who present an algorithm for synthesising probabilistic programs by filling holes in user-defined program sketches. Hutchison [2016] presented a convenient generative grammar for interactive Tabular model creation, which could be adapted to automatic model suggestion.

Finally, the idea of treating functions as expansible macros could be transplanted to other probabilistic languages, such as Stan [Gelman et al., 2015]. There is currently some ongoing work towards this goal [Gorinova, 2017].

Semantics of Probabilistic Programs Our semantics of an untyped probabilistic lambda-calculus only defines distributions on functions treated in a purely syntactic way, and different representations of mathematically equivalent functions are not identified in our semantics. Defining distributions on mathematical functions in higher-order, Turing-complete functional languages with higher-order recursion is still an open problem.

One approach would be to extend the work of Staton et al. [2016] and Heunen et al. [2017] to an untyped lambda-calculus—it is, however, not yet clear how this could be done. Another idea would be to adapt the operational, metric-based approach presented here and design a different metric on terms, quantifying true, behavioural closeness of terms, as suggested by Crubillé and Dal Lago [2017].

Verification of Inference Algorithms As discussed in section 7.5.2, the variant of Metropolis-Hastings used in this dissertation can be inefficient for programs with conditional branches, due to the use of linear traces. An obvious improvement would be the use of a more elaborate and efficient variable naming scheme, for example one based on the labelled lambda calculus—traces would then be maps, rather than lists. This would require redefining the measure space of program traces.

Another important goal would be to prove correctness of a single-site version of MH, updating only one variable, and variables depending on it, at the time. In the case of this variant, the additional difficulty is that the proposal kernel has no density with respect to the stock measure on traces, so a more general theory of MCMC on general state spaces (such as the one presented in [Tierney, 1998]) has to be used. However, Cai [2016] has already presented a generic proof of correctness of such an algorithm, using an abstract representation of a program as a dependent sequence of probability kernels and a map of names of random variables, so it would be enough to instantiate this framework with an appropriate translation of lambda-calculus terms to sequences of kernels and an appropriate variable naming scheme.

It may be interesting to formalise the proof of correctness of MH for probabilistic programs, for example in Isabelle, which has support for measure theory [Hölzl and Heller, 2011].

Furthermore, other sampling-based inference algorithms are increasingly being used in probabilistic programming, most notably Sequential Monte Carlo, which forms the basis of a promising new approach to inference, called inference compilation [Le et al., 2017]. Because of this, it would be very beneficial to prove correctness of SMC.

Appendix A

Alpha-equivalence of Tabular programs

We identify Tabular programs up to consistent renaming of internal column names and local variables occurring in expressions. Following [Pitts, 2013, p. 133], we define alpha-equivalence by means of a variable permutation operator $(x\ y)X$, which naively replaces all (free and bound) occurrences of x in part of syntax X (a schema, table, model, expression or type) with y and vice versa. Note that this operator does not change external column names and table names, as they are considered separate syntactic categories from variables.

We also make use of a function $\text{vars}(X)$, which returns the set of all (free and bound) variables occurring in part of syntax X . This function is formally defined as follows:

Variables (Free and Bound): $\text{vars}(E), \text{vars}(R), \text{vars}(M), \text{vars}(T), \text{vars}(\mathbb{T})$

$$\text{vars}(x) \triangleq \{x\}$$

$$\text{vars}(s) \triangleq \emptyset$$

$$\text{vars}(\text{sizeof}(t)) \triangleq \emptyset$$

$$\text{vars}(g(E_1, \dots, E_n)) \triangleq \text{vars}(E_1) \cup \dots \cup \text{vars}(E_n)$$

$$\text{vars}(\text{if } E \text{ then } F_1 \text{ else } F_2) \triangleq \text{vars}(E) \cup \text{vars}(F_1) \cup \text{vars}(F_2)$$

$$\text{vars}([E_1, \dots, E_n]) \triangleq \text{vars}(E_1) \cup \dots \cup \text{vars}(E_n)$$

$$\text{vars}(E[F]) \triangleq \text{vars}(E) \cup \text{vars}(F)$$

$$\text{vars}([\text{for } x < e \rightarrow F]) \triangleq \{x\} \cup \text{vars}(F)$$

$$\text{vars}(\text{infer}.D[e_1, \dots, e_m].d(E)) = \text{vars}(e_1) \cup \dots \cup \text{vars}(e_m) \cup \text{vars}(E)$$

$$\text{vars}(E : t.c) \triangleq \text{vars}(E)$$

$$\begin{aligned}
\text{vars}(t.c) &\triangleq \emptyset \\
\text{vars}(x.c) &\triangleq \{x\} \\
\text{vars}(\varepsilon) &\triangleq \emptyset \\
\text{vars}(M[e_1 < e_2]) &\triangleq \text{vars}(M) \cup \text{vars}(e_1) \cup \text{vars}(e_2) \\
\text{vars}(\mathbb{T} R) &\triangleq \text{vars}(\mathbb{T}) \cup \text{vars}(R) \\
\text{vars}((c = e) :: R) &\triangleq \text{vars}(e) \cup \text{vars}(R) \\
\text{vars}([]) &\triangleq \emptyset \\
\text{vars}(S ! \text{spc}) &\triangleq \emptyset \\
\text{vars}(\mathbf{mod}(e) ! \text{spc}) &\triangleq \text{vars}(e) \\
\text{vars}(T[e]) &\triangleq \text{vars}(T) \cup \text{vars}(e) \\
\text{vars}([]) &\triangleq \emptyset \\
\text{vars}((c \triangleright x : T \ell \text{ viz } M) :: \mathbb{T}) &\triangleq \{x\} \cup \text{vars}(T) \cup \text{vars}(M) \cup \text{vars}(\mathbb{T})
\end{aligned}$$

We begin by defining alpha-equivalence of basic expressions:

Alpha equivalence for expressions: $E_1 =_\alpha E_2$

<p>(ALPHA EXPRESSION)</p> $\frac{e_1 = e_2}{e_1 =_\alpha e_2}$	<p>(ALPHA Deref STATIC)</p> $\frac{}{t.c =_\alpha t.c}$	<p>(ALPHA Deref INST)</p> $\frac{E =_\alpha E'}{E : t.c =_\alpha E' : t.c}$
<p>(ALPHA RANDOM)</p> $\frac{F_j =_\alpha F'_j \quad \forall j \in 1..n}{D[e_1, \dots, e_m](F_1, \dots, F_n) =_\alpha D[e_1, \dots, e_m](F'_1, \dots, F'_n)}$		
<p>(ALPHA ITER)</p> $\frac{(z x)F =_\alpha (z y)F' \quad z \notin \text{vars}(F, F', x, y)}{[\mathbf{for} \ x < e \rightarrow F] =_\alpha [\mathbf{for} \ y < e \rightarrow F']}$	<p>(ALPHA INDEX)</p> $\frac{E =_\alpha E' \quad F =_\alpha F'}{E[F] =_\alpha E'[F']}$	
<p>(ALPHA INFER)</p> $\frac{E =_\alpha E'}{\mathbf{infer}.D[e_1, \dots, e_m].c_j(E) =_\alpha \mathbf{infer}.D[e_1, \dots, e_m].c_j(E')}$		<p>(ALPHA PRIM)</p> $\frac{E_i =_\alpha E'_i \quad \forall i \in 1..n}{g(E_1, \dots, E_n) =_\alpha g(E'_1, \dots, E'_n)}$
<p>(IF)</p> $\frac{E_1 =_\alpha E'_1 \quad E_2 =_\alpha E'_2 \quad E_3 =_\alpha E'_3}{\mathbf{if} \ E_1 \ \mathbf{then} \ E_2 \ \mathbf{else} \ E_3 =_\alpha \mathbf{if} \ E'_1 \ \mathbf{then} \ E'_2 \ \mathbf{else} \ E'_3}$		<p>(ALPHA ARRAY)</p> $\frac{E_i =_\alpha E'_i \quad \forall i \in 0..n-1}{[E_0, \dots, E_{n-1}] =_\alpha [E'_0, \dots, E'_{n-1}]}$

(ALPHA FUNREF)

$$\frac{}{x.c =_{\alpha} x.c}$$

At the level of primitive expressions, there is only one construct, namely the **for** loop, which binds a variable, so these rules effectively state that two expressions are considered α -equivalent if they are the same up to renaming of variables bound by **for** loops. The (ALPHA ITER) rule states that two loops $[\mathbf{for} \ x < e \rightarrow F]$ and $[\mathbf{for} \ y < e \rightarrow F']$ (with same e) are alpha equivalent if the expressions F and F' , with the bound variables x and y replaced by a single variable fresh in both F and F' , are α -equivalent.

All the other rules are just congruence rules, which state that two expressions are α -equivalent if all of their subexpressions which can contain **for** loops are α -equivalent, and all other components are the same. We do not need α equivalence rules for indexed expressions e —they contain no binders, so two indexed expressions are α -equivalent if and only if they are equal.

Below are the rules for (compound) model expressions:

Alpha equivalence for model expressions: $M_1 =_{\alpha} M_2$

<p>(ALPHAEQ EMPTY)</p> $\frac{}{\varepsilon =_{\alpha} \varepsilon}$	<p>(ALPHAEQ MODELEXPR)</p> $\frac{E_1 =_{\alpha} E_2}{E_1 =_{\alpha} E_2}$
<p>(ALPHAEQ MODELAPPL)</p> $\frac{\mathbb{T}_1 =_{\alpha} \mathbb{T}_2}{\mathbb{T}_1 R =_{\alpha} \mathbb{T}_2 R}$	<p>(ALPHAEQ MODELINDEXED)</p> $\frac{M_1 =_{\alpha} M_2}{M_1[e_1 < e_2] =_{\alpha} M_2[e_1 < e_2]}$

The (ALPHAEQ EMPTY) rule is trivial, and (ALPHAEQ MODELEXPR) just states that two α -equivalent basic expressions are also α -equivalent as model expressions. The (ALPHAEQ MODELAPPL) rule says that two function applications are α -equivalent if the function table themselves are α -equivalent (as defined below), and argument lists are the same (they have no binders, so they are α -equivalent only if they are equal). Finally, (ALPHAEQ MODELINDEXED) states that two indexed models are α equivalent if the underlying models are α equivalent and the index and size expressions are equal (again, they can have no binders).

We define alpha-equivalence for tables as follows:

Alpha equivalence for tables: $\mathbb{T}_1 =_\alpha \mathbb{T}_2$

(ALPHAEQ COLUMN)

$$M_1 =_\alpha M_2 \quad (x \ z) \mathbb{T}_1 =_\alpha (y \ z) \mathbb{T}_2$$

$$\frac{z \notin \text{vars}(x, y, \mathbb{T}_1, \mathbb{T}_2)}{(c \triangleright x : T \ \ell \ \text{viz } M_1) :: \mathbb{T}_1 =_\alpha (c \triangleright y : T \ \ell \ \text{viz } M_2) :: \mathbb{T}_2}$$

(ALPHAEQ EMPTY)

$$\frac{}{[] =_\alpha []}$$

By (ALPHAEQ COLUMN), two non-empty tables are α -equivalent if and only if the model expressions in their first columns are α -equivalent, the external names, levels, visibilities and types in the first columns are the same (recall that column types have no binders, so types are α -equivalent only if they are the same) and the rest of the two tables, with occurrences of the internal names of the first columns replaced by a single fresh variable, are α -equivalent. The (ALPHAEQ EMPTY) rule is trivial.

Finally, the following rules define the α -equivalence of schemas:

Alpha equivalence for schemas: $\mathbb{S}_1 =_\alpha \mathbb{S}_2$

(ALPHA SCHEMA []) (ALPHA SCHEMA TABLE)

$$\mathbb{T} =_\alpha \mathbb{T}' \quad \mathbb{S} =_\alpha \mathbb{S}'$$

$$\frac{}{[] =_\alpha []}$$

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

The (ALPHA SCHEMA TABLE) rule states that two non-empty schemas are α -equivalent if the first tables and the rests of the schemas are α -equivalent and the names of the first tables match (recall that table names are not α -equivalent). The (ALPHA SCHEMA []) rule states that two empty schemas are always α -equivalent.

Appendix B

Proofs of the Propositions from Section 4.4.6

B.1 Proposition 1

As usual, the proof of the proposition requires some additional lemmas:

- Lemma 81 (Weakening)** (1) *If $\Gamma_1, \Gamma_2 \vdash^{pc} E : U$ and $\Gamma_1 \vdash T$ and $x \notin \text{dom}(\Gamma_1, \Gamma_2)$, then $\Gamma_1, x :^\ell T, \Gamma_2 \vdash^{pc} E : U$.*
- (2) *If $\Gamma_1, \Gamma_2 \vdash^{pc} R : Q \rightarrow Q'$ and $\Gamma_1 \vdash T$ and $c \notin \text{dom}(\Gamma_1, \Gamma_2)$ and $x \notin \text{dom}(Q)$, then $\Gamma_1, x :^\ell T, \Gamma_2 \vdash^{pc} R : Q \rightarrow Q'$.*
- (3) *If $\Gamma_1, \Gamma_2 \vdash^{pc} M : Q$ and $\Gamma_1 \vdash T$ and $x \notin \text{dom}(\Gamma_1, \Gamma_2)$ and $x \notin \text{dom}(M)$ then $\Gamma_1, x :^\ell T, \Gamma_2 \vdash^{pc} M : Q$.*
- (4) *If $\Gamma_1, \Gamma_2 \vdash^{pc} \mathbb{T} : Q$ and $\Gamma_1 \vdash T$ and $x \notin \text{dom}(\Gamma_1, \Gamma_2)$ and $x \notin \text{dom}(\mathbb{T})$, then $\Gamma_1, x :^\ell T, \Gamma_2 \vdash^{pc} \mathbb{T} : Q$.*

Proof: 1.) By induction on the derivation of $\Gamma_1, \Gamma_2 \vdash^{pc} E : U$.

2.) By induction on the derivation of $\Gamma_1, \Gamma_2 \vdash^{pc} R : Q \rightarrow Q'$.

3.) and 4.) By simultaneous induction on the derivation of $\Gamma_1, \Gamma_2 \vdash^{pc} M : Q$ and $\Gamma_1, \Gamma_2 \vdash^{pc} \mathbb{T} : Q$

5.) By induction on the derivation of $\Gamma \vdash^{pc} \mathbb{S} : \text{Sty}$ ■

Lemma 82 (Derived judgments) (1) *If $\Gamma \vdash^{pc} E : T$, then $\Gamma \vdash \diamond$*

(2) If $\Gamma \vdash^{pc} M : Q$, then $\Gamma \vdash \diamond$

(3) If $\Gamma \vdash^{pc} \mathbb{T} : Q$, then $\Gamma \vdash \diamond$

(4) If $\Gamma \vdash \mathbb{S} : Sty$, then $\Gamma \vdash \diamond$

Proof: 1.) By induction on the derivation of $\Gamma \vdash^{pc} E : T$.

2.) and 3.) By simultaneous induction on the derivation of $\Gamma \vdash^{pc} M : Q$ and $\Gamma \vdash^{pc} \mathbb{T} : Q$

4.) By induction on the derivation of $\Gamma \vdash^{pc} \mathbb{S} : Sty$ ■

The following sequence of lemmas (Lemmas 83 to 86) shows that all the judgments in the type system are preserved by turning the first column of a Q type into a standalone variable and updating references to it in the rest of the judgment.

Lemma 83 (1) If $\Gamma, x : (c \triangleright y : T \ell viz) :: Q', \Gamma' \vdash \diamond$ and $y \notin \text{dom}(\Gamma) \cup \text{dom}(\Gamma') \cup \{x\}$ and $c \neq \text{ret}$, then $\Gamma, y :^\ell T, x : Q', \Gamma' \vdash \diamond$.

(2) If $\Gamma, x : (c \triangleright y : T \ell viz) :: Q', \Gamma' \vdash T$ and $y \notin \text{dom}(\Gamma) \cup \text{dom}(\Gamma') \cup \{x\}$ and $c \neq \text{ret}$, then $\Gamma, y :^\ell T, x : Q', \Gamma' \vdash T$

(3) If $\Gamma, x : (c \triangleright y : T \ell viz) :: Q', \Gamma' \vdash Q$ and $y \notin \text{dom}(\Gamma) \cup \text{dom}(\Gamma') \cup \{x\}$ and $c \neq \text{ret}$, then $\Gamma, y :^\ell T, x : Q', \Gamma' \vdash Q$

(4) If $\Gamma, x : (c \triangleright y : T \ell viz) :: Q', \Gamma' \vdash^{pc} e : T$ and $y \notin \text{dom}(\Gamma) \cup \text{dom}(\Gamma') \cup \{x\}$ and $c \neq \text{ret}$, then $\Gamma, y :^\ell T, x : Q', \Gamma' \vdash^{pc} e : T$.

Proof: By simultaneous induction on the derivation of $\Gamma, x : (c \triangleright y : T \ell viz) :: Q', \Gamma' \vdash \diamond$; $\Gamma, x : (c \triangleright y : T \ell viz) :: Q', \Gamma' \vdash T$; $\Gamma, x : (c \triangleright y : T \ell viz) :: Q', \Gamma' \vdash Q$ and $\Gamma, x : (c \triangleright y : T \ell viz) :: Q', \Gamma' \vdash^{pc} e : T$. Interesting cases:

- Case (ENV FUN):

If $\Gamma' \neq \emptyset$, then the result follows immediately by the induction hypothesis. Now, let us assume that $\Gamma' = \emptyset$:

(ENV FUN) (red(Q))

$\Gamma \vdash (c \triangleright y : T \ell viz) :: Q' \quad x \notin \text{dom}(\Gamma)$

$\Gamma, x : (c \triangleright y : T \ell viz) :: Q' \vdash \diamond$

By a derived judgment, we have $\Gamma \vdash \diamond$. Now we need to split on viz :

- Subcase $viz = \mathbf{output}$: In this case, $\Gamma \vdash (c \triangleright y : T \ell viz) :: Q'$ must have been derived by (TABLE TYPE OUTPUT), so we have $\Gamma \vdash T$ and $\Gamma, y :^\ell T \vdash Q'$. By (ENV FUN), we get $\Gamma, y :^\ell T, x : Q' \vdash \diamond$, as required.
- Subcase $viz = \mathbf{input}$: similar.

- Case (INDEX VAR):

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

$$\frac{\Gamma, x : (c \triangleright y : T \ell viz) :: Q', \Gamma' \vdash \diamond \quad \Gamma, x : (c \triangleright y : T \ell viz) :: Q', \Gamma' = \Gamma'_1, z :^\ell T, \Gamma''_1}{\Gamma, x : (c \triangleright y : T \ell viz) :: Q', \Gamma' \vdash^{pc} z : T}$$

By induction hypothesis, $\Gamma, y :^\ell T, x : Q', \Gamma' \vdash \diamond$. Since the well-formedness of the environment ensures that $x \neq z$, we have $z \in \text{dom}(\Gamma)$ or $z \in \text{dom}(\Gamma')$, and so $\Gamma, y :^\ell T, x : Q', \Gamma' = \Gamma'_2, z :^\ell T, \Gamma''_2$. Hence, $\Gamma, y :^\ell T, x : Q', \Gamma' \vdash^{pc} z : T$ by (INDEX VAR).

- Case (FUNREFRET):

(FUNREFRET)

$$\frac{\Gamma, x : (c \triangleright y : T \ell viz) :: Q', \Gamma' \vdash \diamond \quad \Gamma, x : (c \triangleright y : T \ell viz) :: Q', \Gamma' = \Gamma'_1, z : Q_z, \Gamma''_1 \quad Q_z = Q'_z @ [(ret \triangleright y : U \ell' \mathbf{output})] \quad \ell' \leq pc}{\Gamma, x : (c \triangleright y : T \ell viz) :: Q', \Gamma' \vdash^{pc} z : U}$$

By induction hypothesis, $\Gamma, y :^\ell T, x : Q', \Gamma' \vdash \diamond$. Now, we need to consider two cases: $z \neq x$ and $z = x$.

If $z \neq x$, we must have $\Gamma, y :^\ell T, x : Q', \Gamma' = \Gamma'_2, z : Q_z, \Gamma''_2$ for some Γ'_2, Γ''_2 . Thus, (FUNREFRET) gives $\Gamma, x : (c \triangleright y : T \ell viz) :: Q', \Gamma' \vdash^{pc} z : U$, as required.

If $z = x$, then we have $Q_z = (c \triangleright y : T \ell viz) :: Q'$, where $Q' = Q'' @ [(ret \triangleright y : U \ell' \mathbf{output})]$ for some Q'' , because $c \neq \text{ret}$.

Thus, we get $\Gamma, y :^\ell T, x : Q', \Gamma' \vdash^{pc} x : U$ by (FUNREFRET), as required. ■

Lemma 84 *If $\Gamma, x : (c \triangleright y : T \ell viz) :: Q', \Gamma' \vdash T <: U$ and $y \notin \text{dom}(\Gamma) \cup \text{dom}(\Gamma') \cup \{x\}$ and $c \neq \text{ret}$, then $\Gamma, y :^\ell T, x : Q', \Gamma' \vdash T <: U$.*

Proof: By induction on the derivation of $\Gamma, x : (c \triangleright y : T \ell viz) :: Q', \Gamma' \vdash T <: U$, with appeal to Lemma 83. ■

Lemma 85 *If $\Gamma, x : (c \triangleright y : T \ell \text{ viz}) :: Q', \Gamma' \vdash^{pc} E : U$ and $y \notin \text{dom}(\Gamma) \cup \text{dom}(\Gamma') \cup \{x\}$ and $c \neq \text{ret}$, then $\Gamma, y :^\ell T, x : Q', \Gamma' \vdash^{pc} E\langle y/x.c \rangle : U$.*

Proof: By induction on the derivation of $\Gamma, x : (c \triangleright y : T \ell \text{ viz}) :: Q', \Gamma' \vdash^{pc} E : U$.

Interesting cases:

- Case (FUNREF): $E = z.d$

In all the following subcases, we have $\Gamma, y :^\ell T, x : Q', \Gamma' \vdash \diamond$ by Lemma 83.

- If $z \neq x$, then $z.d\langle y/x.c \rangle = z.d$ and the derivation was of the form

(FUNREF)

$$\begin{array}{c} \Gamma_1 \vdash \diamond \quad \Gamma_1 = \Gamma'_1, z : Q_1, \Gamma''_1 \\ Q_1 = Q'_1 @ (d \triangleright z' : U \ell' \text{ viz}) @ Q''_1 \\ \ell' \leq pc \\ \hline \Gamma_1 \vdash^{pc} z.d : U \end{array}$$

where $\Gamma_1 = \Gamma, x : (c \triangleright y : T \ell \text{ viz}) :: Q', \Gamma'$. Since $x \neq z$ implies $z \in \text{dom}(\Gamma)$ or $z \in \text{dom}(\Gamma')$, we have $\Gamma, y :^\ell T, x : Q', \Gamma' = \Gamma'_2, z : Q_1, \Gamma''_2$. Thus, $\Gamma, y :^\ell T, x : Q', \Gamma' \vdash^{pc} z.d : U$ holds by (FUNREF).

- If $z = x$ and $d \neq c$, then $x.d\langle y/x.c \rangle = x.d$, $u = T$ and the derivation was of the form

(FUNREF)

$$\begin{array}{c} \Gamma_1 \vdash \diamond \quad \Gamma_1 = \Gamma, x : (c \triangleright y : T \ell \text{ viz}) :: Q', \Gamma' \\ Q' = Q_1 @ (d \triangleright z' : U \ell' \text{ viz}) @ Q''_1 \\ \ell' \leq pc \\ \hline \Gamma_1 \vdash^{pc} x.d : U \end{array}$$

We obviously get $\Gamma, y :^\ell T, x : Q', \Gamma' \vdash^{pc} x.d : U$ by applying (FUNREF) again.

- If $z = x$ and $d = c$, then $x.c\langle y/x.c \rangle = y$ and the derivation was

(FUNREF)

$$\begin{array}{c} \Gamma_1 \vdash \diamond \\ \Gamma_1 = \Gamma', x : ((c \triangleright y : T \ell \text{ viz}) :: Q'), \Gamma'' \\ \ell \leq pc \\ \hline \Gamma_1 \vdash^{pc} x.c : T \end{array}$$

We have $\Gamma, y :^\ell T, x : Q', \Gamma' \vdash^{pc} y : T$ by (INDEX VAR) and (INDEX EXPRESSION).

- Case:

(ITER) (where $z \notin \text{fv}(T)$)

$\Gamma, x : (c \triangleright y : T \ell \text{ viz}) :: Q', \Gamma' \vdash^{\text{static}} e : \text{int} ! \text{det}$

$\Gamma, x : (c \triangleright y : T \ell \text{ viz}) :: Q', \Gamma', z : {}^{pc}(\text{mod}(e) ! \text{det}) \vdash^{pc} F : U'$

$\Gamma, x : (c \triangleright y : T \ell \text{ viz}) :: Q', \Gamma' \vdash^{pc} [\text{for } z < e \rightarrow F] : U'[e]$

By induction hypothesis, $\Gamma, y : {}^\ell T, x : Q', \Gamma' \vdash^{\text{static}} e : \text{int} ! \text{det}$ and $\Gamma, y : {}^\ell T, x : Q', \Gamma', z : {}^{pc}(\text{mod}(e) ! \text{det}) \vdash^{pc} F \langle y/x.c \rangle : U'$. Hence, by (ITER), $\Gamma, y : {}^\ell T, x : Q', \Gamma' \vdash^{pc} [\text{for } z < e \rightarrow F \langle y/x.c \rangle] : U'[e]$ as required

- Case:

(SUBSUM)

$\Gamma, x : (c \triangleright y : T \ell \text{ viz}) :: Q' \vdash^{pc} E : U' \quad \Gamma, x : (c \triangleright y : T \ell \text{ viz}) :: Q' \vdash U' <: U$

$\Gamma, x : (c \triangleright y : T \ell \text{ viz}) :: Q' \vdash^{pc} E : U$

By induction hypothesis, $\Gamma, y : {}^\ell T, x : Q' \vdash^{pc} E \langle y/x.c \rangle : U'$. By Lemma 84, we have $\Gamma, y : {}^\ell T, x : Q' \vdash U' <: U$. Hence, by (SUBSUM), $\Gamma, y : {}^\ell T, x : Q' \vdash^{pc} E \langle y/x.c \rangle : U$.

■

Lemma 86 *If $\Gamma, x : (c \triangleright y : T \ell \text{ viz}) :: Q', \Gamma' \vdash^{pc} R : Q \rightarrow Q'$ and $y \notin \text{dom}(\Gamma) \cup \{x\}$ and $c \neq \text{ret}$, then $\Gamma, y : {}^\ell T, x : Q', \Gamma' \vdash^{pc} R : Q \rightarrow Q'$.*

Proof: This is a straightforward induction on the derivation of $\Gamma, x : (c \triangleright y : T \ell \text{ viz}) :: Q', \Gamma' \vdash^{pc} R : Q \rightarrow Q'$. Note that $x.c$ cannot appear in R, Q nor Q' . ■

Lemma 87 (1) *If $\Gamma, x : (c \triangleright y : T \ell \text{ viz}) :: Q', \Gamma' \vdash^{pc} M : Q$ and $y \notin \text{dom}(\Gamma) \cup \text{dom}(\Gamma') \cup \{x\}$ and $c \neq \text{ret}$, then $\Gamma, y : {}^\ell T, x : Q', \Gamma' \vdash^{pc} M \langle y/x.c \rangle : Q$.*

(2) *If $\Gamma, x : (c \triangleright y : T \ell \text{ viz}) :: Q', \Gamma' \vdash^{pc} \mathbb{T} : Q$ and $y \notin \text{dom}(\Gamma) \cup \text{dom}(\Gamma') \cup \{x\}$ and $c \neq \text{ret}$, then $\Gamma, y : {}^\ell T, x : Q', \Gamma' \vdash^{pc} \mathbb{T} \langle y/x.c \rangle : Q$.*

Proof: By simultaneous induction on the derivation of $\Gamma, x : (c \triangleright y : T \ell \text{ viz}) :: Q', \Gamma' \vdash^{pc} M : Q$ and $\Gamma, x : (c \triangleright y : T \ell \text{ viz}) :: Q', \Gamma' \vdash^{pc} \mathbb{T} : Q$.

- Case:

(TABLE \square)

$\Gamma \vdash \diamond$

$\Gamma \vdash^{pc} \square : \square$

Trivial.

- Case:

(TABLE CORE OUTPUT)

$$\begin{array}{l} \Gamma, x : (c \triangleright y : T \ell \text{ viz}) :: Q', \Gamma' \vdash^{\ell' \wedge pc} E : T \\ \Gamma, x : (c \triangleright y : T \ell \text{ viz}) :: Q', \Gamma', z : \ell' \wedge pc \ T' \vdash^{pc} \mathbb{T}_1 : Q_1 \\ c \notin \text{names}(Q_1) \end{array}$$

$$\Gamma, x : (c \triangleright y : T \ell \text{ viz}) :: Q', \Gamma' \vdash^{pc} (c \triangleright z : T' \ell' \text{ output } E) :: \mathbb{T}_1 : (c \triangleright z : T' (\ell' \wedge pc) \text{ output}) @ Q_1$$

By Lemma 85, we have $\Gamma, y : \ell \ T, x : Q', \Gamma' \vdash^{\ell' \wedge pc} E \langle y/x.c \rangle : T$. By induction hypothesis, $\Gamma, y : \ell \ T, x : Q', \Gamma', z : \ell' \wedge pc \ T' \vdash^{pc} \mathbb{T}_1 \langle y/x.c \rangle : Q_1$. Hence, by (TABLE CORE OUTPUT), we get $\Gamma, y : \ell \ T, x : Q', \Gamma' \vdash^{pc} (c \triangleright z : T' (\ell' \wedge pc) \text{ output } E \langle y/x.c \rangle) :: \mathbb{T}_1 \langle y/x.c \rangle : (c \triangleright z : T' \ell' \text{ output}) :: Q_1$, as required.

- Cases (TABLE CORE LOCAL) and (TABLE INPUT) similar.

- Case:

(MODEL APPL)

$$\begin{array}{l} \Gamma, x : (c \triangleright y : T \ell \text{ viz}) :: Q', \Gamma' \vdash^{pc} \mathbb{T} : Q^* \quad \text{fun}(\mathbb{T}) \\ \Gamma, x : (c \triangleright y : T \ell \text{ viz}) :: Q', \Gamma' \vdash^{pc} R : Q^* \rightarrow Q \end{array}$$

$$\Gamma, x : (c \triangleright y : T \ell \text{ viz}) :: Q', \Gamma' \vdash^{pc} \mathbb{T} R : Q$$

By induction hypothesis, $\Gamma, y : \ell \ T, x : Q', \Gamma' \vdash^{pc} \mathbb{T} \langle y/x.c \rangle : Q^*$ and by Lemma 86, $\Gamma, y : \ell \ T, x : Q', \Gamma' \vdash^{pc} R : Q^* \rightarrow Q$. Hence, by (MODEL APPL), we get $\Gamma, y : \ell \ T, x : Q', \Gamma' \vdash^{pc} \mathbb{T} \langle y/x.c \rangle R : Q$ as required.

- Case:

(MODEL INDEXED)

$$\begin{array}{l} \Gamma, x : (c \triangleright y : T \ell \text{ viz}) :: Q', \Gamma' \vdash^{pc} M : Q^* \\ \Gamma, x : (c \triangleright y : T \ell \text{ viz}) :: Q', \Gamma' \vdash^{pc} e_{\text{index}} : \text{mod}(e_{\text{size}}) ! \text{rnd} \end{array}$$

$$\Gamma, x : (c \triangleright y : T \ell \text{ viz}) :: Q', \Gamma' \vdash^{pc} M[e_{\text{index}} < e_{\text{size}}] : Q^*[e_{\text{size}}]$$

By induction hypothesis, $\Gamma, y : \ell \ T, x : Q', \Gamma' \vdash^{pc} M \langle y/x.c \rangle : Q^*$ and $\Gamma, y : \ell \ T, x : Q', \Gamma' \vdash^{pc} e_{\text{index}} : \text{mod}(e_{\text{size}}) ! \text{rnd}$. Thus, by (MODEL INDEXED), $\Gamma, y : \ell \ T, x : Q', \Gamma' \vdash^{pc} M \langle y/x.c \rangle [e_{\text{index}} < e_{\text{size}}] : Q^*[e_{\text{size}}]$ as required.

- Case:

(TABLE OUTPUT)

$$\Gamma, x : (c \triangleright y : T \ell \text{ viz}) :: Q', \Gamma' \vdash^{\ell' \wedge pc} M : Q_c$$

$$\Gamma, x : (c \triangleright y : T \ell \text{ viz}) :: Q', \Gamma', z : Q_c \vdash^{pc} \mathbb{T} : Q_1$$

$$Q_c = Q'_c @ [(\text{ret} \triangleright y : T \ell' \text{ output})]$$

$$\text{names}(c.Q_c) \cap \text{names}(Q_1) = \emptyset$$

$$\Gamma, x : (c \triangleright y : T \ell \text{ viz}) :: Q', \Gamma' \vdash^{pc} (d \triangleright z : T' \ell' \text{ output } M) :: \mathbb{T} : (c.Q_c) @ Q_1$$

By induction hypothesis, $\Gamma, y :^\ell T, x : Q', \Gamma' \vdash^{\ell' \wedge pc} M \langle y/x.c \rangle : Q_c$ and $\Gamma, y :^\ell T, x : Q', \Gamma', z : Q_c \vdash^{pc} \mathbb{T} \langle y/x.c \rangle : Q_1$. Hence, by (TABLE OUTPUT), $\Gamma, y :^\ell T, x : Q', \Gamma' \vdash^{pc} (d \triangleright z : T' \ell' \text{ output } M) \langle y/x.c \rangle :: \mathbb{T} \langle y/x.c \rangle : (c.Q_c) @ Q_1$.

- Case (TABLE LOCAL) similar.

■

In order to prove that indexing function tables preserves typing, it is convenient to define a new operator $\text{index_env}(\Gamma, A, e)$, which takes a typing environment Γ , a set A of variable names and an integer-valued indexed expression e , and returns the environment Γ with the type T of every variable in A changed to the array type $T[e]$.

The reason for this is that a model expression in an indexed column is expected to be well-typed not in the same environment as in the original table, but in an environment we would obtain after typechecking the previous columns of the indexed table. If we keep track of the names of indexed columns by adding them to A , we can use index_env to obtain the correct, modified environment.

Environment indexing: $\text{index_env}(\Gamma, A, e)$

$$\text{index_env}(\emptyset, A, e) = \emptyset$$

$$\text{index_env}(\Gamma, x : Q, A, e) = \text{index_env}(\Gamma, A, e), x : Q$$

$$\text{index_env}(\Gamma, t : Q, A, e) = \text{index_env}(\Gamma, A, e), t : Q$$

$$\text{index_env}(\Gamma, x :^\ell T, A, e) = \begin{cases} \text{index_env}(\Gamma, A \setminus \{x\}, e), x :^\ell T[e] & \text{if } x \in A \\ \text{index_env}(\Gamma, A, e), x :^\ell T & \text{otherwise} \end{cases}$$

We also need a conformance relation relating the set A of names of indexed columns to the environment Γ . This relation ensures that only **static rnd** columns can appear in A , reflecting the fact that only the expressions in such columns can be turned into arrays.

Variable set conformance: $\Gamma \vdash A$

(EMPTY)	(NOTINA)	(INA)	(TABLE)
$\frac{}{\emptyset \vdash A}$	$\frac{\Gamma \vdash A \quad x \notin A}{\Gamma, x : {}^\ell T \vdash A}$	$\frac{\Gamma \vdash A \quad \mathbf{rnd}(T)}{\Gamma, x : \mathbf{static} T \vdash A \cup \{x\}}$	$\frac{\Gamma \vdash A}{\Gamma, t : Q \vdash A}$
(FUN)			
$\frac{\Gamma \vdash A \quad x \notin A}{\Gamma, x : Q \vdash A}$			

Note that we do not require all **static rnd** variables in Γ to appear in A , because if we were not to index some of these columns, well-typedness would still be preserved.

Lemma 88 *If $\Gamma, x : {}^\ell T, \Gamma' \vdash A$ and $x \in A$, then $\ell = \mathbf{static}$ and $\mathbf{rnd}(T)$.*

Proof: Trivial. ■

The following sequence of lemmas (Lemmas 89, 91, 92) show that all judgments in the type system are preserved by indexing.

Lemma 89 (1) *If $\Gamma, \Gamma', \Gamma'' \vdash \diamond$ and $\Gamma' \vdash A$ and $\Gamma \vdash \mathbf{static} e : \mathbf{int} ! \mathbf{det}$, then*

$$\Gamma, \text{index_env}(\Gamma', A, e), \Gamma'' \vdash \diamond.$$

(2) *If $\Gamma, \Gamma', \Gamma'' \vdash T$ and $\Gamma' \vdash A$ and $\Gamma \vdash \mathbf{static} e : \mathbf{int} ! \mathbf{det}$ then $\Gamma, \text{index_env}(\Gamma', A, e), \Gamma'' \vdash T$*

(3) *If $\Gamma, \Gamma', \Gamma'' \vdash Q$ and $\Gamma' \vdash A$ and $\Gamma \vdash \mathbf{static} e : \mathbf{int} ! \mathbf{det}$ then $\Gamma, \text{index_env}(\Gamma', A, e), \Gamma'' \vdash Q$*

(4) *If $\Gamma, \Gamma', \Gamma'' \vdash \mathbf{static} e' : T$ and $\mathbf{det}(T)$ and $\Gamma' \vdash A$ and $\Gamma \vdash \mathbf{static} e : \mathbf{int} ! \mathbf{det}$, then*

$$\Gamma, \text{index_env}(\Gamma', A, e), \Gamma'' \vdash \mathbf{static} e' : T.$$

Proof: By simultaneous induction on the derivation of $\Gamma, \Gamma', \Gamma'' \vdash \diamond$, $\Gamma, \Gamma', \Gamma'' \vdash T$, $\Gamma, \Gamma', \Gamma'' \vdash Q$ and $\Gamma, \Gamma', \Gamma'' \vdash \mathbf{static} e' : T$. Interesting cases:

- Case (TYPE ARRAY):

(TYPE ARRAY)

$$\frac{\Gamma, \Gamma', \Gamma'' \vdash U \quad \Gamma, \Gamma', \Gamma'' \vdash \mathbf{static} e' : \mathbf{int} ! \mathbf{det}}{\Gamma, \Gamma', \Gamma'' \vdash U[e']}$$

By induction hypothesis, $\Gamma, \text{index_env}(\Gamma', A, e), \Gamma'' \vdash U$ and $\Gamma, \text{index_env}(\Gamma', A, e), \Gamma'' \vdash^{\text{static}} e' : \text{int} ! \text{det}$, so we have $\Gamma, \text{index_env}(\Gamma', A, e), \Gamma'' \vdash U[e']$ by (TYPE ARRAY).

- Case (INDEX VAR):

(INDEX VAR)

$$\frac{\Gamma, \Gamma', \Gamma'' \vdash \diamond \quad \Gamma, \Gamma', \Gamma'' = \Gamma_1, x :^{\text{static}} T, \Gamma_2}{\Gamma, \Gamma', \Gamma'' \vdash^{\text{static}} x : T}$$

By induction hypothesis, $\Gamma, \text{index_env}(\Gamma', A, e), \Gamma'' \vdash \diamond$. We have either $x \in \text{dom}(\Gamma) \cup \text{dom}(\Gamma'')$ or $x \in \text{dom}(\Gamma')$. In the former case, $\Gamma, \text{index_env}(\Gamma', A, e), \Gamma'' \vdash^{\text{static}} x : T$ follows trivially. In the latter, we have $\Gamma' = \Gamma'_1, x :^{\text{static}} T, \Gamma'_2$. Since $\text{det}(T)$ by assumption, Lemma 88 implies $x \notin A$, so $\text{index_env}(\Gamma', A, e) = \Gamma''_1, x :^{\text{static}} T, \Gamma''_2$. Thus, we have $\Gamma, \text{index_env}(\Gamma', A, e), \Gamma'' \vdash^{\text{static}} x : T$ by (INDEX VAR).

- Case (TABLE TYPE OUTPUT):

(TABLE TYPE OUTPUT)

$$\frac{\Gamma, \Gamma', \Gamma'' \vdash T \quad \Gamma, \Gamma', \Gamma'', x :^\ell T \vdash Q' \quad c \notin \text{names}(Q')}{\Gamma, \Gamma', \Gamma'' \vdash (c \triangleright x : T \ell \text{ output}) :: Q'}$$

By induction hypothesis, $\Gamma, \text{index_env}(\Gamma', A, e), \Gamma'' \vdash T$ and $\Gamma, \text{index_env}(\Gamma', A, e), \Gamma'', x :^\ell T \vdash Q'$, so by (TABLE TYPE OUTPUT) we get $\Gamma, \text{index_env}(\Gamma', A, e), \Gamma'' \vdash (c \triangleright x : T \ell \text{ output}) :: Q'$.

- Case (ENV VAR):

- Subcase $\Gamma'' \neq \emptyset$: we have $\Gamma'' = \Gamma^*, x :^{pc} T$.

(ENV VAR)

$$\frac{\Gamma, \Gamma', \Gamma^* \vdash T \quad x \notin \text{dom}(\Gamma) \cup \text{dom}(\Gamma') \cup \text{dom}(\Gamma^*)}{\Gamma, \Gamma', \Gamma^*, x :^{pc} T \vdash \diamond}$$

By induction hypothesis, $\Gamma, \text{index_env}(\Gamma', A, e), \Gamma^* \vdash T$. Since indexing preserves the domain of environment, we have $x \notin \text{dom}(\Gamma) \cup \text{dom}(\text{index_env}(\Gamma', A, e)) \cup \text{dom}(\Gamma^*)$. Hence, $\Gamma, \text{index_env}(\Gamma', A, e), \Gamma^*, x :^{pc} T \vdash \diamond$ by (ENV VAR).

- Subcase $\Gamma'' = \emptyset, \Gamma' \neq \emptyset$: we have $\Gamma' = \Gamma^{**}, x :^{pc} T$.

(ENV VAR)

$$\frac{\Gamma, \Gamma^{**} \vdash T \quad x \notin \text{dom}(\Gamma) \cup \text{dom}(\Gamma^{**})}{\Gamma, \Gamma^{**}, x :^{pc} T \vdash \diamond}$$

If $x \notin A$, then $\text{index_env}(\Gamma^{**}, x :^{pc} T, A, e) = \text{index_env}(\Gamma^{**}, A, e), x :^{pc} T$ and $\Gamma^{**} \vdash A$. By induction hypothesis, $\Gamma, \text{index_env}(\Gamma^{**}, A, e) \vdash T$. Since indexing preserves the domain of environment, we have

$x \notin \text{dom}(\Gamma) \cup \text{dom}(\text{index_env}(\Gamma^{**}, x :^{pc} T, A, e))$. Thus, $\Gamma, \text{index_env}(\Gamma^{**}, x :^{pc} T, A, e) \vdash \diamond$ follows by (ENV VAR).

If $x \in A$, then $\text{index_env}(\Gamma^{**}, x :^{pc} T, A, e) = \text{index_env}(\Gamma^{**}, A \setminus \{x\}, e), x :^{pc} T[e]$ and $\Gamma^{**} \vdash A \setminus \{x\}$. By induction hypothesis, $\Gamma, \text{index_env}(\Gamma^{**}, A \setminus \{x\}, e) \vdash T$. By weakening, $\Gamma, \text{index_env}(\Gamma^{**}, A \setminus \{x\}, e) \vdash^{\text{static}} e : \text{int} ! \text{det}$, and so by (TYPE ARRAY), $\Gamma, \text{index_env}(\Gamma^{**}, A \setminus \{x\}, e) \vdash T[e]$. Hence, by (ENV VAR), $\Gamma, \text{index_env}(\Gamma^{**}, x :^{pc} T, A, e) \vdash \diamond$.

– Subcase $\Gamma'' = \emptyset, \Gamma' = \emptyset$: trivial.

- Case (FUNREFRET):

(FUNREFRET)

$$\frac{\Gamma, \Gamma', \Gamma'' \vdash \diamond \quad \Gamma, \Gamma', \Gamma'' = \Gamma'_1, x : Q, \Gamma''_1 \quad Q = Q' @ (\text{ret} \triangleright y : U \text{ static output})}{\Gamma, \Gamma', \Gamma'' \vdash^{\text{static}} x : U}$$

By induction hypothesis, $\Gamma, \text{index_env}(\Gamma', A, e), \Gamma'' \vdash \diamond$. If $x \in \text{dom}(\Gamma) \cup \text{dom}(\Gamma'')$, then the result follows immediately. If $x \in \text{dom}(\Gamma')$, then we have $\Gamma, \text{index_env}(\Gamma', A, e), \Gamma'' = \Gamma'_2, x : Q, \Gamma''_2$ for some Γ'_2 and Γ''_2 , because indexing an environment does not change Q -types. Hence, we have $\Gamma, \text{index_env}(\Gamma', A, e), \Gamma'' \vdash^{\text{static}} x : U$, as required.

■

Lemma 90 For all $D_{\text{spc}} : [x_1 : T_1, \dots, x_m : T_m](y_1 : U_1, \dots, y_n : U_n) \rightarrow T$, $\text{det}(T_i)$ for every $i \in 1..m$ and $\text{rnd}(T)$.

Proof: By inspection.

■

Lemma 91 If $\Gamma, \Gamma', \Gamma'' \vdash T <: U$ and $\Gamma' \vdash A$ and $\Gamma \vdash^{\text{static}} e : \text{int} ! \text{det}$, then $\Gamma, \text{index_env}(\Gamma', A, e), \Gamma'' \vdash T <: U$.

Proof: By induction on the derivation of $\Gamma, \Gamma', \Gamma'' \vdash T <: U$, with appeal to Lemma 89.

■

Lemma 92 *If $\Gamma, \Gamma' \vdash^{\ell \wedge pc} E : T$ and $\Gamma' \vdash A$ and $\Gamma \vdash^{pc} e_1 : \mathbf{mod}(e_2) ! \mathbf{rnd}$ and $A \cap \text{dom}(\Gamma) = \emptyset$ and either $\mathbf{det}(T)$ or $\ell = \mathbf{inst}$, then $\Gamma, \text{index_env}(\Gamma', A, e_2) \vdash^{\ell \wedge pc} E[A, e_1] : T$.*

Proof: By induction on the derivation of $\Gamma, \Gamma' \vdash^{\ell \wedge pc} E : T$. Interesting cases:

- Case (INDEX VAR):

$$\begin{array}{c} \text{(INDEX VAR) (for } \ell' \leq \ell \wedge pc) \\ \Gamma, \Gamma' \vdash \diamond \quad \Gamma, \Gamma' = \Gamma_1, x :^{\ell'} T, \Gamma_2 \\ \hline \Gamma, \Gamma' \vdash^{\ell \wedge pc} x : T \end{array}$$

By Lemma 89, we have $\Gamma, \text{index_env}(\Gamma', A, e_2) \vdash \diamond$. We know that either $x \in \text{dom}(\Gamma)$ or $x \in \text{dom}(\Gamma')$. If $x \in \text{dom}(\Gamma)$, then $x \notin A$, so $x[A, e_1] = x$ and we trivially get $\Gamma, \text{index_env}(\Gamma', A, e_2) \vdash^{\ell \wedge pc} x : T$. Now, let us assume that $x \in \text{dom}(\Gamma')$.

If $x \notin A$, then again $x[A, e_1] = x$ and $\text{index_env}(\Gamma', A, e_2) = \Gamma'_1, x :^{\ell'} T, \Gamma'_2$, so obviously $\Gamma, \text{index_env}(\Gamma', A, e_2) \vdash^{\ell \wedge pc} x : T$ by (INDEX VAR).

If $x \in A$, then $x[A, e_1] = x[e_1]$, $\text{index_env}(\Gamma', A, e_2) = \Gamma'_1, x :^{\ell'} T[e_2], \Gamma'_2$ and by Lemma 88, $\ell' = \mathbf{static}$ and $\mathbf{rnd}(T)$, which by assumption implies $\ell = \mathbf{inst}$ (and so $\ell \wedge pc = pc$). By (INDEX VAR), we have $\Gamma, \text{index_env}(\Gamma', A, e_2) \vdash^{pc} x : T[e_2]$. By weakening, $\Gamma, \text{index_env}(\Gamma', A, e_2) \vdash^{pc} e_1 : \mathbf{mod}(e_2) ! \mathbf{rnd}$. Thus, by (INDEX), $\Gamma, \text{index_env}(\Gamma', A, e_2) \vdash^{pc} x[e_1] : T$.

- Case (INDEX):

$$\begin{array}{c} \text{(INDEX)} \\ \text{space}(T) \leq \text{spc} \\ \Gamma, \Gamma' \vdash^{\ell \wedge pc} E' : T[e'] \quad \Gamma, \Gamma' \vdash^{\ell \wedge pc} F : \mathbf{mod}(e') ! \text{spc} \\ \hline \Gamma, \Gamma' \vdash^{\ell \wedge pc} E'[F] : T \vee \text{spc} \end{array}$$

If $\mathbf{det}(T \vee \text{spc})$, then $\mathbf{det}(T)$ and $\text{spc} = \mathbf{det}$, so we can apply the induction hypothesis to both assumptions. This yields $\Gamma, \text{index_env}(\Gamma', A, e_2) \vdash^{\ell \wedge pc} E'[A, e_1] : T[e']$ and $\Gamma, \text{index_env}(\Gamma', A, e_2) \vdash^{\ell \wedge pc} F[A, e_1] : \mathbf{mod}(e') ! \text{spc}$. Hence, we get $\Gamma, \text{index_env}(\Gamma', A, e_2) \vdash^{\ell \wedge pc} (E'[F])[A, e_1] : T \vee \text{spc}$ by (INDEX).

- Case (ITER):

$$\begin{array}{c} \text{(ITER) (where } x \notin \text{fv}(T)) \\ \Gamma, \Gamma' \vdash^{\mathbf{static}} e : \mathbf{int} ! \mathbf{det} \\ \Gamma, \Gamma', x :^{\ell \wedge pc} (\mathbf{mod}(e) ! \mathbf{det}) \vdash^{\ell \wedge pc} F : T \\ \hline \Gamma, \Gamma' \vdash^{\ell \wedge pc} [\mathbf{for } x < e \rightarrow F] : T[e] \end{array}$$

We have $\text{index_env}(\Gamma', x : \ell \wedge pc \text{ (mod}(e) ! \mathbf{det}) , A, e_2) = \text{index_env}(\Gamma', A, e_2), x : \ell \wedge pc \text{ (mod}(e) ! \mathbf{det})$. By Lemma 89, $\Gamma, \text{index_env}(\Gamma', A, e_2) \vdash^{\mathbf{static}} e : \mathbf{int} ! \mathbf{det}$ and by induction hypothesis, $\Gamma, \text{index_env}(\Gamma', A, e_2), x : \ell \wedge pc \text{ (mod}(e) ! \mathbf{det}) \vdash^{\ell \wedge pc} F[A, e_1] : T$. Thus, we get the required result directly by (ITER).

- Case (FUNREFRET):

(FUNREFRET)

$$\Gamma, \Gamma' \vdash \diamond \quad \Gamma, \Gamma' = \Gamma'_1, x : Q, \Gamma''_1$$

$$Q = Q' @ (\text{ret} \triangleright y : T \ell' \text{ output})$$

$$\ell' \leq \ell \wedge pc$$

$$\Gamma, \Gamma' \vdash^{\ell \wedge pc} x : T$$

Whether $x \in \text{dom}(\Gamma)$ or $x \in \text{dom}(\Gamma')$, we have $\Gamma, \text{index_env}(\Gamma', A, e_2) = \Gamma'_2, x : Q, \Gamma''_2$, so we have $\Gamma, \text{index_env}(\Gamma', A, e_2) \vdash^{\ell \wedge pc} x : T$ by (FUNREFRET), which is what we needed to show, since $\Gamma' \vdash A$ and $\text{dom}(\Gamma) \cap A = \emptyset$ imply $x \notin A$, and so $x[A, e_1] = x$.

- Case (RANDOM):

(RANDOM) (where $\sigma(U) \triangleq U \{e_1/x_1\} \dots \{e_m/x_m\}$)

$$D_{\mathbf{rnd}} : [x_1 : T_1, \dots, x_m : T_m](c_1 : U_1, \dots, c_n : U_n) \rightarrow T$$

$$\Gamma, \Gamma' \vdash^{\mathbf{static}} e_i : T_i \quad \forall i \in 1..m \quad \Gamma, \Gamma' \vdash^{\ell \wedge pc} F_j : \sigma(U_j) \quad \forall j \in 1..n \quad \Gamma, \Gamma' \vdash \diamond$$

$$\{x_1, \dots, x_m\} \cap (\bigcup_i \text{fv}(e_i)) = \emptyset \quad x_i \neq x_j \text{ for } i \neq j$$

$$\Gamma, \Gamma' \vdash^{\ell \wedge pc} D[e_1, \dots, e_m](F_1, \dots, F_n) : \sigma(T)$$

By Lemma 90, $\mathbf{det}(T_i)$ for every $i \in 1..m$ and $\mathbf{rnd}(T)$, so $\ell = \mathbf{inst}$ and Lemma 89 implies that $\Gamma, \text{index_env}(\Gamma', A, e_2) \vdash \diamond$ and $\Gamma, \text{index_env}(\Gamma', A, e_2) \vdash^{\mathbf{static}} e_i : T_i$ for all $i \in 1..m$. By induction hypothesis, $\Gamma, \text{index_env}(\Gamma', A, e_2) \vdash^{\ell \wedge pc} F_j : \sigma(U_j)$ for all $j \in 1..n$. Thus, by (RANDOM), $\Gamma, \text{index_env}(\Gamma', A, e_2) \vdash^{pc} D[e_1, \dots, e_m](F_1, \dots, F_n) : \sigma(T)$

■

The following properties (Lemmas 93, 94, lemma:swap-type) show that we can swap two independent variables in the environment in typing judgments for expressions.

Lemma 93 (1) If $\Gamma, x : \ell T, y : \ell' U, \Gamma' \vdash \diamond$ and $\Gamma \vdash U$, then $\Gamma, y : \ell' U, x : \ell T, \Gamma \vdash \diamond$

(2) If $\Gamma, x : \ell T, y : \ell' U, \Gamma' \vdash V$ and $\Gamma \vdash U$, then $\Gamma, y : \ell' U, x : \ell T, \Gamma \vdash V$

(3) If $\Gamma, x :^\ell T, y :^{\ell'} U, \Gamma' \vdash Q$ and $\Gamma \vdash U$, then $\Gamma, y :^{\ell'} U, x :^\ell T, \Gamma' \vdash Q$

(4) If $\Gamma, x :^\ell T, y :^{\ell'} U, \Gamma' \vdash^{pc} e : V$ and $\Gamma \vdash U$, then $\Gamma, y :^{\ell'} U, x :^\ell T, \Gamma' \vdash^{pc} e : V$

Proof: By simultaneous induction on the derivation of $\Gamma, x :^\ell T, y :^{\ell'} U, \Gamma' \vdash \diamond$; $\Gamma, x :^\ell T, y :^{\ell'} U, \Gamma' \vdash V$; $\Gamma, x :^\ell T, y :^{\ell'} U, \Gamma' \vdash Q$ and $\Gamma, x :^\ell T, y :^{\ell'} U, \Gamma' \vdash^{pc} e : V$ ■

Lemma 94 If $\Gamma, x :^\ell T, y :^{\ell'} U, \Gamma' \vdash V_1 <: V_2$ and $\Gamma \vdash U$, then $\Gamma, y :^{\ell'} U, x :^\ell T, \Gamma' \vdash V_1 <: V_2$

Proof: By induction on the derivation of $\Gamma, x :^\ell T, y :^{\ell'} U, \Gamma' \vdash V_1 <: V_2$ with appeal to Lemma 93. ■

Lemma 95 If $\Gamma, x :^\ell T, y :^{\ell'} U, \Gamma' \vdash^{pc} E : V$ and $\Gamma \vdash U$, then $\Gamma, y :^{\ell'} U, x :^\ell T, \Gamma' \vdash^{pc} E : V$

Proof: By induction on the derivation of $\Gamma, x :^\ell T, y :^{\ell'} U, \Gamma' \vdash^{pc} E : V$, with appeal to lemmas 93 and 94. ■

Lemma 96 If $\Gamma, \Gamma' \vdash^{\text{static}} E : T$ and $\Gamma' \vdash A$ and $\Gamma \vdash^{\text{static}} e_2 : \text{int} ! \text{det}$ and $A \cap \text{dom}(\Gamma) = \emptyset$ and $i \notin \text{dom}(\Gamma) \cup \text{dom}(\Gamma')$ then $\Gamma, \text{index_env}(\Gamma', A, e_2), i :^{\text{static}} \text{mod}(e_2) ! \text{det} \vdash^{\text{static}} E[A, i] : T$.

Proof: By induction on the derivation of $\Gamma, \Gamma' \vdash^{\text{static}} E : T$. Interesting cases:

- Case (INDEX VAR):

(INDEX VAR)

$$\frac{\Gamma, \Gamma' \vdash \diamond \quad \Gamma, \Gamma' = \Gamma_1, x :^{\text{static}} T, \Gamma_2}{\Gamma, \Gamma' \vdash^{\text{static}} x : T}$$

By Lemma 89, $\Gamma, \text{index_env}(\Gamma', A, e_2) \vdash \diamond$. If $x \in \text{dom}(\Gamma)$, then $x \notin A$, so $x[A, i] = x$ and so we get $\Gamma, \text{index_env}(\Gamma', A, e_2), i :^{\text{static}} \text{mod}(e_2) ! \text{det} \vdash^{\text{static}} x : T$ by (INDEX VAR) and weakening. Now, let us assume that $x \in \text{dom}(\Gamma')$.

If $x \notin A$, then again $x[A, i] = x$ and $\text{index_env}(\Gamma', A, e_2) = \Gamma'_1, x :^{\text{static}} T, \Gamma'_2$, so obviously $\Gamma, \text{index_env}(\Gamma', A, e_2), i :^{\text{static}} \text{mod}(e_2) ! \text{det} \vdash^{\text{static}} x : T$ by (INDEX VAR) and weakening.

If $x \in A$, then $x[A, i] = x[i]$ and $\text{index_env}(\Gamma', A, e_2) = \Gamma'_1, x :^{\text{static}} T[e_2], \Gamma'_2$. By (INDEX VAR) and weakening, we have $\Gamma, \text{index_env}(\Gamma', A, e_2), i :^{\text{static}} \mathbf{mod}(e_2) ! \mathbf{det} \vdash^{\text{static}} x : T[e_2]$. Thus, by (INDEX), $\Gamma, \text{index_env}(\Gamma', A, e_2), i :^{\text{static}} \mathbf{mod}(e_2) ! \mathbf{det} \vdash^{\text{static}} x[i] : T$.

- Case (ITER):

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

$\Gamma, \Gamma' \vdash^{\text{static}} e : \mathbf{int} ! \mathbf{det}$

$\Gamma, \Gamma', x :^{\text{static}} (\mathbf{mod}(e) ! \mathbf{det}) \vdash^{\text{static}} F : T$

$\Gamma, \Gamma' \vdash^{\text{static}} [\mathbf{for } x < e \rightarrow F] : T[e]$

By Lemma 89, $\Gamma, \text{index_env}(\Gamma', A, e_2) \vdash^{\text{static}} e : \mathbf{int} ! \mathbf{det}$ and by induction hypothesis, $\Gamma, \text{index_env}(\Gamma', A, e_2), x :^{\text{static}} (\mathbf{mod}(e) ! \mathbf{det}), i :^{\text{static}} \mathbf{mod}(e_2) ! \mathbf{det} \vdash^{\ell \wedge pc} F[A, i] : T$. Since $\Gamma \vdash \mathbf{mod}(e_2) ! \mathbf{det}$, by Lemma 95 we have $\Gamma, \text{index_env}(\Gamma', A, e_2), i :^{\text{static}} \mathbf{mod}(e_2) ! \mathbf{det}, x :^{\text{static}} (\mathbf{mod}(e) ! \mathbf{det}) \vdash^{\ell \wedge pc} F[A, i] : T$. By weakening, $\Gamma, \text{index_env}(\Gamma', A, e_2), i :^{\text{static}} \mathbf{mod}(e_2) ! \mathbf{det} \vdash^{\text{static}} e : \mathbf{int} ! \mathbf{det}$. Thus, by (ITER), we get $\Gamma, \text{index_env}(\Gamma', A, e_2), i :^{\text{static}} \mathbf{mod}(e_2) ! \mathbf{det} \vdash^{\text{static}} [\mathbf{for } x < e \rightarrow F[A, i]] : T[e]$, as required.

- Case (FUNREFRET):

(FUNREFRET)

$\Gamma, \Gamma' \vdash \diamond \quad \Gamma, \Gamma' = \Gamma'_1, x : Q, \Gamma''_1$

$Q = Q' @ (\mathbf{ret} \triangleright y : T \mathbf{static output})$

$\Gamma, \Gamma' \vdash^{\text{static}} x : T$

If $x \in \text{dom}(\Gamma)$, the result is obvious. Otherwise, if $x \in \text{dom}(\Gamma')$, from $\Gamma' \vdash A$ we can infer that $x \notin A$, so $\text{index_env}(\Gamma', A, e_2) = \Gamma'_2, x : Q, \Gamma''_2$ and $x[A, i] = x$. Hence, by (FUNREFRET) and weakening, $\Gamma, \text{index_env}(\Gamma', A, e_2), i :^{\text{static}} \mathbf{mod}(e_2) \vdash^{\text{static}} x : T$, as required.

■

Lemma 97 *If $\Gamma, \Gamma' \vdash^{pc} \mathbb{T} : Q$ and $\text{NoQry}(\mathbb{T})$ and $\Gamma \vdash^{pc} e_1 : \mathbf{mod}(e_2) ! \mathbf{rnd}$ and $\text{fv}(e_1) \cup \text{fv}(e_2) \notin A$ and $\Gamma' \vdash A$ and $A \cap \text{dom}(\Gamma) = \emptyset$, then $\Gamma, \text{index_env}(\Gamma', A, e_2) \vdash^{pc} \text{index}_A(\mathbb{T}, e_1, e_2) : Q[e_2]$.*

Proof: By induction on the derivation of $\Gamma, \Gamma' \vdash^{pc} \mathbb{T} : Q$

- Case:

(TABLE \square)

$$\frac{\Gamma, \Gamma' \vdash \diamond}{\Gamma, \Gamma' \vdash^{pc} \square : \square}$$

By Lemma 89. we have $\Gamma, \text{index_env}(\Gamma', A, e) \vdash \diamond$, so $\Gamma, \text{index_env}(\Gamma', A, e) \vdash^{pc} \square : \square$ by (TABLE \square).

- Case:

(TABLE INPUT)

$$\frac{\Gamma, \Gamma', x :^{\ell \wedge pc} T \vdash^{pc} \mathbb{T}' : Q' \quad c \notin \text{names}(Q')}{\Gamma, \Gamma' \vdash^{pc} (c \triangleright x : T \ell \text{ input } \varepsilon) :: \mathbb{T}' : (c \triangleright x : T (\ell \wedge pc) \text{ input}) :: Q'}$$

Assume w.l.o.g. that x is fresh, that is, $x \notin A$ (x is bound, so we can alpha-convert it if needed). Then we have $\Gamma', x :^{\ell \wedge pc} T \vdash A$ and $x \notin \text{dom}(\text{index_env}(\Gamma', A, e), x :^{\ell \wedge pc} T)$. By induction hypothesis, $\Gamma, \text{index_env}(\Gamma', A, e_2), x :^{\ell \wedge pc} T \vdash^{pc} \text{index}_A(\mathbb{T}', e_1, e_2) : Q'[e_2]$. Thus, by (TABLE INPUT), $\Gamma, \text{index_env}(\Gamma, A, e_2) \vdash^{pc} (c \triangleright x : T \ell \text{ input } \varepsilon) :: \text{index}_A(\mathbb{T}', e_1, e_2) : (c \triangleright x : T (\ell \wedge pc) \text{ input}) :: Q'[e_2]$, as required.

- Case:

(TABLE CORE OUTPUT)

$$\frac{\Gamma, \Gamma' \vdash^{\ell \wedge pc} E : T \quad \Gamma, \Gamma', x :^{\ell \wedge pc} T \vdash^{pc} \mathbb{T} : Q \quad c \notin \text{names}(Q')}{\Gamma, \Gamma' \vdash^{pc} (c \triangleright x : T \ell \text{ output } E) :: \mathbb{T} : (c \triangleright x : T (\ell \wedge pc) \text{ output}) :: Q}$$

- Subcase: **det**(T) or $\ell = \text{inst}$:

By induction hypothesis, $\Gamma, \text{index_env}(\Gamma, A, e_2), x :^{\ell \wedge pc} T \vdash^{pc} \text{index}_A(\mathbb{T}, e_1, e_2) : Q[e_2]$. By Lemma 92, $\Gamma, \text{index_env}(\Gamma, A, e_2) \vdash^{\ell \wedge pc} E[A, e_1] : T$. Thus, since indexing a Q -type preserves its external names, by (TABLE CORE OUTPUT) we get $\Gamma, \text{index_env}(\Gamma', A, e_2) \vdash^{pc} (c \triangleright x : T \ell \text{ output } E[A, e_1]) :: \text{index}_A(\mathbb{T}, e_1, e_2) : (c \triangleright x : T (\ell \wedge pc) \text{ output}) :: Q[e_2]$, as required.

- Subcase: **rnd**(T), $\ell = \text{static}$:

We have $\Gamma', x :^{\ell \wedge pc} T \vdash A \cup \{x\}$ and obviously $A \cup \{x\} \cap \Gamma = \emptyset$, so by induction hypothesis, $\Gamma, \text{index_env}(\Gamma', A, e_2), x :^{\ell \wedge pc} T[e_2] \vdash^{pc} \text{index}_{A \cup \{x\}}(\mathbb{T}, e_1, e_2) : Q$. By Lemma 96, $\Gamma, \text{index_env}(\Gamma', A, e_2) \vdash^{\ell \wedge pc} [\text{for } i < e_2 \rightarrow E[A, i]] : T[e_2]$. Hence, by (TABLE CORE OUTPUT),

$\Gamma, \text{index_env}(\Gamma', A, e_2) \vdash^{pc} (c \triangleright x : T[e_2] \ell \text{ output } [\text{for } i < e_2 \rightarrow E[A, i]]) :: \text{index}_A(\mathbb{T}, e_1, e_2) : (c \triangleright x : T[e_2] (\ell \wedge pc) \text{ output}) :: Q[e_2]$ as required.

- Case:

(TABLE CORE LOCAL) (where $x \notin \text{fv}(Q)$)

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

Similar to (TABLE CORE OUTPUT).

■

Corollary 5 *If $\Gamma \vdash^{pc} \mathbb{T} : Q$ and $\text{fun}(\mathbb{T})$ and $\text{NoQry}(\mathbb{T})$ and $\Gamma \vdash^{pc} e_1 : \text{mod}(e_2) ! \text{rnd}$, then $\Gamma \vdash^{pc} \text{index}_{\emptyset}(\mathbb{T}, e_1, e_2) : Q[e_2]$.*

Lemma 98 *If $\Gamma \vdash^{pc} R : Q \rightarrow Q'$ and $\Gamma \vdash^{\text{static}} e_{\text{size}} : \text{int} ! \text{det}$ then $\Gamma \vdash^{pc} R : Q[e_{\text{size}}] \rightarrow Q'[e_{\text{size}}]$.*

Proof: By induction on the derivation of $\Gamma \vdash^{pc} R : Q \rightarrow Q'$:

- Case:

(ARG INPUT)

$$\frac{\Gamma \vdash^{\ell \wedge pc} e : T \quad \Gamma \vdash^{pc} R : Q\{e/x\} \rightarrow Q'}{\Gamma \vdash^{pc} ((c = e) :: R) : ((c \triangleright x : T \ell \text{ input}) :: Q) \rightarrow Q'}$$

By induction hypothesis, $\Gamma \vdash^{pc} R : Q[e_{\text{size}}]\{e/x\} \rightarrow Q'[e_{\text{size}}]$. Hence, by (ARG INPUT), we have $\Gamma \vdash^{pc} ((c = e) :: R) : ((c \triangleright x : T \ell \text{ input}) :: Q[e_{\text{size}}]) \rightarrow Q'[e_{\text{size}}]$, as required.

- Case:

(ARG OUTPUT)

$$\frac{\Gamma, x : \ell \wedge pc \vdash^{pc} R : Q \rightarrow Q' \quad c \neq \text{ret} \quad x \notin \text{fv}(R)}{\Gamma \vdash^{pc} R : ((c \triangleright x : T \ell \text{ output}) :: Q) \rightarrow ((c \triangleright x : T (\ell \wedge pc) \text{ output}) :: Q')}$$

- Subcase $\text{det}(T)$:

By weakening, $\Gamma, x : \ell \wedge pc \vdash^{\text{static}} e_{\text{size}} : \text{int} ! \text{det}$. By induction hypothesis, $\Gamma, x : \ell \wedge pc \vdash^{pc} R : Q[e_{\text{size}}] \rightarrow Q'[e_{\text{size}}]$. By (ARG OUTPUT), we have $\Gamma \vdash^{pc} R : ((c \triangleright x : T \ell \text{ output}) :: Q[e_{\text{size}}]) \rightarrow ((c \triangleright x : T (\ell \wedge pc) \text{ output}) :: Q'[e_{\text{size}}])$, as required.

- Subcase $\text{rnd}(T)$:

By type well-formedness rules, we can show that $\text{rnd}(T)$ implies $x \notin \text{fv}(Q) \cup \text{fv}(Q')$. Thus, it is straightforward to show that $\Gamma, x : \ell \wedge pc \vdash^{pc} T[e_{\text{size}}] \vdash^{pc} R :$

$Q \rightarrow Q'$. Thus, by (ARG OUTPUT), we have $\Gamma \vdash^{pc} R : ((c \triangleright x : T[e_{size}] \ell \text{ output}) :: Q[e_{size}]) \rightarrow ((c \triangleright x : T[e_{size}] (\ell \wedge pc) \text{ output}) :: Q'[e_{size}])$ as required

• Case:

(ARG RET)

$\Gamma \vdash T$

$\Gamma \vdash^{pc} R : (\text{ret} \triangleright x : T \ell \text{ output}) \rightarrow (\text{ret} \triangleright x : T (\ell \wedge pc) \text{ output})$

– Subcase **det**(T):

Trivial.

– Subcase **rnd**(T):

Since from $\Gamma \vdash T$ and $\Gamma \vdash^{\text{static}} e_{size} : \text{int} ! \text{det}$, we can derive $\Gamma \vdash T[e_{size}]$, the result $\Gamma \vdash^{pc} R : (\text{ret} \triangleright x : T[e_{size}] \ell \text{ output}) \rightarrow (\text{ret} \triangleright x : T[e_{size}] (\ell \wedge pc) \text{ output})$ follows immediately by (ARG RET). ■

Lemma 99 *If $\Gamma \vdash^{\ell} E : T$ and $\ell \leq \ell'$, then $\Gamma \vdash^{\ell'} E : T$*

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

Lemma 100 (1) *If $\Gamma, x :^{\ell \wedge pc} T, \Gamma' \vdash \diamond$ and $\Gamma \vdash^{\ell \wedge pc} e : T$, then*

$\Gamma, (\Gamma' \{e/x\}) \vdash \diamond$.

(2) *If $\Gamma, x :^{\ell \wedge pc} T, \Gamma' \vdash T$ and $\Gamma \vdash^{\ell \wedge pc} e : T$, then $\Gamma, (\Gamma' \{e/x\}) \vdash T \{e/x\}$*

(3) *If $\Gamma, x :^{\ell \wedge pc} T, \Gamma' \vdash Q$ and $\Gamma \vdash^{\ell \wedge pc} e : T$, then $\Gamma, (\Gamma' \{e/x\}) \vdash Q \{e/x\}$*

(4) *If $\Gamma, x :^{\ell \wedge pc} T, \Gamma' \vdash^{pc} e' : U$ and $\Gamma \vdash^{\ell \wedge pc} e : T$, then*

$\Gamma, (\Gamma' \{e/x\}) \vdash^{pc} e' \{e/x\} : U \{e/x\}$.

Proof: By simultaneous induction on the derivation of $\Gamma, x :^{\ell \wedge pc} T, \Gamma' \vdash \diamond$, $\Gamma, x :^{\ell \wedge pc} T, \Gamma' \vdash T$, $\Gamma, x :^{\ell \wedge pc} T, \Gamma' \vdash Q$ and $\Gamma, x :^{\ell \wedge pc} T, \Gamma' \vdash^{pc} e' : U$. Interesting cases:

- Case (INDEX VAR):

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

$$\frac{\Gamma, x :^{\ell \wedge pc} T, \Gamma' \vdash \diamond \quad \Gamma, x :^{\ell \wedge pc} T, \Gamma' = \Gamma_1, y :^{\ell'} U, \Gamma_2}{\Gamma, x :^{\ell \wedge pc} T, \Gamma' \vdash^{pc} y : U}$$

By induction hypothesis, $\Gamma, (\Gamma' \{e/x\}) \vdash \diamond$. If $y \neq x$, the result is obvious. If $y = x$, then we have $T = U$. Since it follows from the derived judgments that $x \notin \text{fv}(U)$, we have $U = U \{e/x\}$. Hence, $\Gamma \vdash^{\ell \wedge pc} e : U \{e/x\}$, and so by Lemma 99 and weakening, $\Gamma, (\Gamma' \{e/x\}) \vdash^{pc} e : U \{e/x\}$, as required.

- Case (FUNREFRET):

(FUNREFRET)

$$\frac{\begin{array}{l} \Gamma, x :^{\ell \wedge pc} T, \Gamma' \vdash \diamond \quad \Gamma, x :^{\ell \wedge pc} T, \Gamma' = \Gamma'_1, z : Q, \Gamma''_1 \\ Q = Q' @[(\text{ret} \triangleright y : U \ell' \text{ output})] \\ \ell \leq pc \end{array}}{\Gamma, x :^{\ell \wedge pc} T, \Gamma' \vdash^{pc} z : U}$$

By induction hypothesis, $\Gamma, (\Gamma' \{e/x\}) \vdash \diamond$. If $z \in \text{dom}(\Gamma)$, the result follows immediately. If $z \in \text{dom}(\Gamma')$, we have $\Gamma, (\Gamma' \{e/x\}) = \Gamma, \Gamma'_2 \{e/x\}, z : Q \{e/x\}, \Gamma''_2 \{e/x\}$ where $Q \{e/x\} = Q \{e/x\} @[(\text{ret} \triangleright y : U \{e/x\} \ell' \text{ output})]$ (recall that the **fun** predicate does not allow U to contain any variable bound by columns in Q). Hence, by (FUNREFRET), we have $\Gamma, (\Gamma' \{e/x\}) \vdash^{pc} z : U \{e/x\}$

■

Lemma 101 *If $\Gamma, x :^{\ell \wedge pc} T, \Gamma' \vdash^{pc} E : U$ and $\Gamma \vdash^{\ell \wedge pc} e : T$, then $\Gamma, (\Gamma' \{e/x\}) \vdash^{pc} E \{e/x\} : U \{e/x\}$.*

Proof: By induction on the derivation of $\Gamma, x :^{\ell \wedge pc} T, \Gamma' \vdash^{pc} E : T$. Interesting case:

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

$$\frac{\begin{array}{l} \Gamma, x :^{\ell \wedge pc} T, \Gamma' \vdash^{\text{static}} e' : \text{int} ! \text{det} \\ \Gamma, x :^{\ell \wedge pc} T, \Gamma', y :^{pc} (\text{mod}(e') ! \text{det}) \vdash^{pc} F : V \end{array}}{\Gamma, x :^{\ell \wedge pc} T, \Gamma' \vdash^{pc} [\text{for } y < e' \rightarrow F] : V[e']}$$

By Lemma 100, $\Gamma, (\Gamma' \{e/x\}) \vdash^{pc} e' \{e/x\} : \text{int} ! \text{det}$ and by induction hypothesis, $\Gamma, (\Gamma' \{e/x\}), y :^{pc} (\text{mod}(e' \{e/x\}) ! \text{det}) \vdash^{pc} F \{e/x\} : V \{e/x\}$, so we have

$\Gamma, (\Gamma' \{e/x\}) \vdash^{pc} [\text{for } y < e' \{e/x\} \rightarrow F \{e/x\}] : V \{e/x\} [e' \{e/x\}]$ by (ITER).

■

Lemma 102 *If $\Gamma, x :^{\ell \wedge pc} T \vdash^{pc} R : Q \rightarrow Q'$ and $\Gamma \vdash^{\ell \wedge pc} e : T$, then $\Gamma \vdash^{pc} R\{e/x\} : Q\{e/x\} \rightarrow Q'\{e/x\}$.*

Proof: By induction on the derivation of $\Gamma, x :^{\ell \wedge pc} T \vdash^{pc} R : Q \rightarrow Q'$. ■

Lemma 103 (1) *If $\Gamma, x :^{\ell \wedge pc} T \vdash^{pc} M : Q$ and $\Gamma \vdash^{\ell \wedge pc} e : T$, then $\Gamma \vdash^{pc} M\{e/x\} : Q\{e/x\}$.*

(2) *If $\Gamma, x :^{\ell \wedge pc} T \vdash^{pc} \mathbb{T} : Q$ and $\Gamma \vdash^{\ell \wedge pc} e : T$, then $\Gamma \vdash^{pc} \mathbb{T}\{e/x\} : Q\{e/x\}$.*

Proof: By simultaneous induction on the derivation of $\Gamma, x :^{\ell \wedge pc} T \vdash^{pc} M : Q$ and $\Gamma, x :^{\ell \wedge pc} T \vdash^{pc} \mathbb{T} : Q$, with appeal to Lemmas 101 and 102. ■

Lemma 104 (1) *If $\Gamma, x : [(\text{ret} \triangleright y : T \ell \text{ output})], \Gamma' \vdash \diamond$ and $\Gamma \vdash^{\ell \wedge pc} e : T$, then*

$\Gamma, x :^{\ell} T \vdash \diamond$.

(2) *If $\Gamma, x : [(\text{ret} \triangleright y : T \ell \text{ output})], \Gamma' \vdash T$ and $\Gamma \vdash^{\ell \wedge pc} e : T$, then $\Gamma, x :^{\ell} T \vdash T$*

(3) *If $\Gamma, x : [(\text{ret} \triangleright y : T \ell \text{ output})], \Gamma' \vdash Q$ and $\Gamma \vdash^{\ell \wedge pc} e : T$, then $\Gamma, x :^{\ell} T \vdash Q$*

(4) *If $\Gamma, x : [(\text{ret} \triangleright y : T \ell \text{ output})], \Gamma' \vdash^{pc} e' : T$ and $\Gamma \vdash^{\ell \wedge pc} e : T$, then*

$\Gamma, x :^{\ell} T \vdash^{pc} e : T$.

Proof: By simultaneous induction on the derivation of $\Gamma, x : [(\text{ret} \triangleright y : T \ell \text{ output})], \Gamma' \vdash \diamond$; $\Gamma, x : [(\text{ret} \triangleright y : T \ell \text{ output})], \Gamma' \vdash T$; $\Gamma, x : [(\text{ret} \triangleright y : T \ell \text{ output})], \Gamma' \vdash Q$ and $\Gamma, x : [(\text{ret} \triangleright y : T \ell \text{ output})], \Gamma' \vdash Q$. ■

Lemma 105 *If $\Gamma, x : [(\text{ret} \triangleright y : T \ell \text{ output})], \Gamma' \vdash T <: U$ and $\Gamma \vdash^{\ell \wedge pc} e : T$, then $\Gamma, x :^{\ell} T \vdash T <: U$.*

Proof: By induction on the derivation of $\Gamma, x : [(\text{ret} \triangleright y : T \ell \text{ output})], \Gamma' \vdash T <: U$, with appeal to Lemma 104. ■

Lemma 106 *If $\Gamma, x : [(\text{ret} \triangleright y : T \ell \text{ output})], \Gamma' \vdash^{pc} E : U$ then $\Gamma, x : {}^\ell T \vdash^{pc} E : U$.*

Proof: By induction on the derivation of $\Gamma, x : [(\text{ret} \triangleright y : T \ell \text{ output})], \Gamma' \vdash^{pc} E : U$.

Interesting cases:

- Case (FUNREFRET):

(FUNREFRET)

$\Gamma, x : [(\text{ret} \triangleright y : T \ell \text{ output})], \Gamma' \vdash \diamond$

$\Gamma, x : [(\text{ret} \triangleright y : T \ell \text{ output})], \Gamma' = \Gamma'_1, z : Q, \Gamma''_1$

$Q = Q' @ [(\text{ret} \triangleright y' : T' \ell' \text{ output})]$

$\ell' \leq pc$

$\Gamma \vdash^{pc} z : T'$

By Lemma 104, we have $\Gamma, x : {}^\ell T, \Gamma' \vdash \diamond$. If $x \neq z$, the proof is straightforward. Now let us assume $x = z$. Obviously, this implies $Q' = []$ and $[(\text{ret} \triangleright y' : T' \ell' \text{ output})] = [(\text{ret} \triangleright y : T \ell \text{ output})]$. By (INDEX VAR), we have $\Gamma, x : {}^\ell T, \Gamma' \vdash^{pc} x : T$, as required.

- Case (ITER):

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

$\Gamma, x : [(\text{ret} \triangleright y : T \ell \text{ output})], \Gamma' \vdash^{\text{static}} e' : \text{int} ! \text{det}$

$\Gamma, x : [(\text{ret} \triangleright y : T \ell \text{ output})], z : {}^{pc} (\text{mod}(e') ! \text{det}) \vdash^{pc} F : V$

$\Gamma, x : [(\text{ret} \triangleright y : T \ell \text{ output})], \Gamma' \vdash^{pc} [\text{for } z < e' \rightarrow F] : V[e']$

By Lemma 104, $\Gamma, x : {}^\ell T, \Gamma' \vdash^{\text{static}} e' : \text{int} ! \text{det}$ and by induction hypothesis, $\Gamma, x : {}^\ell T, \Gamma', z : {}^{pc} (\text{mod}(e') ! \text{det}) \vdash^{pc} F : V$. Hence, we get the required result by (ITER).

■

Lemma 107 *If $\Gamma, x : [(\text{ret} \triangleright y : T \ell \text{ output})], \Gamma' \vdash^{pc} R : Q \rightarrow Q'$ then $\Gamma, x : {}^\ell T \vdash^{pc} R : Q \rightarrow Q'$.*

Proof: By induction on the derivation of $\Gamma, x : [(\text{ret} \triangleright y : T \ell \text{ output})], \Gamma' \vdash^{pc} R : Q \rightarrow Q'$

■

Lemma 108 (1) *If $\Gamma, x : [(\text{ret} \triangleright y : T \ell \text{ output})], \Gamma' \vdash^{pc} M : Q$ then $\Gamma, x : {}^\ell T \vdash^{pc} M : Q$.*

(2) *If $\Gamma, x : [(\text{ret} \triangleright y : T \ell \text{ output})], \Gamma' \vdash^{pc} \mathbb{T} : Q$ then $\Gamma, x : {}^\ell T \vdash^{pc} \mathbb{T} : Q$.*

Proof: By simultaneous induction on the derivation of $\Gamma, x : [(\text{ret} \triangleright y : T \ell \text{ output})], \Gamma' \vdash^{pc} M : Q$ and $\Gamma, x : [(\text{ret} \triangleright y : T \ell \text{ output})], \Gamma' \vdash^{pc} \mathbb{T} : Q$. ■

Lemma 109 *If $\Gamma \vdash^{pc} R : Q \rightarrow Q'$, then $\text{names}(Q') \subseteq \text{names}(Q)$.*

Proof: By induction on the derivation of $\Gamma \vdash^{pc} R : Q \rightarrow Q'$. ■

Lemma 110 *If $c \notin \text{names}(Q)$ then for all o , $o.c \notin \text{names}(o.Q)$.*

Proof: Obviously, $o.c \neq \text{ret}$, and every element of $\text{names}(o.Q)$ other than ret is of the form $o.d$ for some $d \in \text{names}(Q)$. Thus, $o.c \in \text{names}(o.Q)$ implies $o.c = o.d$, and so $c = d$, which contradicts the assumption. ■

Restatement of Proposition 1 [Type preservation]

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

Proof: By simultaneous induction on the derivation of $M \rightarrow M'$, $\mathbb{T} \rightarrow \mathbb{T}'$ and $\mathbb{S} \rightarrow \mathbb{S}'$.

- Case (RED APPL OUTPUT):

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

$y \notin \text{fv}(\mathbb{T}', R) \cup \{x\} \quad c \neq \text{ret}$

$(o \triangleright x : T \ell \text{ viz } ((c \triangleright y : T' \ell' \text{ output } E) :: \mathbb{T}_f) R) :: \mathbb{T}' \rightarrow$
 $(o.c \triangleright y : T' \ell \wedge \ell' \text{ viz } E) :: (o \triangleright x : T \ell \text{ viz } \mathbb{T}_f R) :: \mathbb{T}' \langle y/x.c \rangle$

- Subcase $\text{viz} = \text{output}$:

In this case, $\Gamma \vdash^{\text{inst}} \mathbb{T} : Q$ must have been derived with:

(TABLE OUTPUT)

- (1) $\Gamma \vdash^\ell (c \triangleright y : T' \ell' \text{ output } E) :: \mathbb{T}_f R : (c \triangleright y : T' (\ell \wedge \ell') \text{ output}) :: Q_f$
- (2) $\Gamma, x : (c \triangleright y : T' (\ell \wedge \ell') \text{ output}) :: Q_f \vdash^{\text{inst}} \mathbb{T}_1 : Q_1$
- (3) $Q_f = Q'_f @ [(\text{ret} \triangleright y : T \ell'' \text{ output})]$
- (4) $(o.c \cup \text{names}(o.Q_f)) \cap \text{names}(Q_1) = \emptyset$

$\Gamma \vdash^{\text{inst}} (o \triangleright x : T \ell \text{ output } ((c \triangleright y : T' \ell' \text{ output } E) :: \mathbb{T}_f) R) :: \mathbb{T}_1 :$

$(o.c \triangleright y : T' (\ell \wedge \ell') \text{ output}) :: (o.Q_f) @ Q_1$

Where $Q = (c \triangleright y : T' \ell' \text{ output}) :: (o.Q_f)@Q_1$ and the first assumption is derived with:

(MODEL APPL)

$$(5) \Gamma \vdash^\ell (c \triangleright y : T' \ell' \text{ output } E) :: \mathbb{T}_f : (c \triangleright y : T' (\ell \wedge \ell') \text{ output}) :: Q^*$$

$$(6) \text{fun}((c \triangleright y : T' \ell' \text{ output } E) :: \mathbb{T}_f)$$

$$(7) \Gamma \vdash^\ell R : (c \triangleright y : T' (\ell \wedge \ell') \text{ output}) :: Q^* \rightarrow (c \triangleright y : T' (\ell \wedge \ell') \text{ output}) :: Q_f$$

$$\Gamma \vdash^\ell (c \triangleright y : T' \ell' \text{ output } E) :: \mathbb{T}_f R : (c \triangleright y : T' (\ell \wedge \ell') \text{ output}) :: Q_f$$

The first and third assumptions of the above must have been derived with

(TABLE CORE OUTPUT)

$$(8) \Gamma \vdash^{\ell \wedge \ell'} E : T' \quad (9) \Gamma, y : \ell \wedge \ell' T' \vdash^\ell \mathbb{T}_f : Q^* \quad (10) c \notin \text{names}(Q^*)$$

$$\Gamma \vdash^\ell (c \triangleright y : T' \ell' \text{ output } E) :: \mathbb{T}_f : (c \triangleright y : T' (\ell \wedge \ell') \text{ output}) :: Q^*$$

and

(ARG OUTPUT)

$$(11) \Gamma, y : \ell \wedge \ell' T' \vdash^\ell R : Q^* \rightarrow Q_f \quad (12) c \neq \text{ret} \quad (13) y \notin \text{fv } R$$

$$\Gamma \vdash^\ell R : ((c \triangleright y : T' (\ell \wedge \ell') \text{ output}) :: Q^*) \rightarrow ((c \triangleright y : T' (\ell \wedge \ell') \text{ output}) :: Q_f)$$

respectively.

By 109 and 110, we have $o.c \notin \text{names}(o.Q_f)$, which combined with assumption (4) gives $o.c \notin \text{names}(o.Q_f@Q_1)$.

By applying (MODEL APPL) to assumptions (9), (11) and (6) (which obviously implies $\text{fun}(\mathbb{T}_f)$) we get $\Gamma, y : \ell \wedge \ell' T' \vdash^\ell \mathbb{T}_f R : Q_f$.

By Lemma 87, we have $\Gamma, y : \ell \wedge \ell' T', x : Q_c \vdash^{\text{inst}} \mathbb{T}_1 \langle y/x.c \rangle : Q_1$.

By (TABLE OUTPUT), the two above results and assumption (3) yield $\Gamma, y : \ell \wedge \ell' T' \vdash^{\text{inst}} (o \triangleright x : T \ell \text{ output } (\mathbb{T}_f R)) :: \mathbb{T}_1 \langle y/x.c \rangle : (o.Q_f)@Q_1$.

Finally, by applying (TABLE CORE OUTPUT) to the above, assumption (8) and $o.c \notin \text{names}(o.Q_f@Q_1)$, we get $\Gamma \vdash^{\text{inst}} (o.c \triangleright y : T' \ell \wedge \ell' \text{ output } E) :: (o \triangleright x : T \ell \text{ output } \mathbb{T}_f R) :: \mathbb{T}' \langle y/x.c \rangle : (o.c \triangleright y : T' (\ell \wedge \ell') \text{ output}) :: (o.Q_f)@Q_1$ as required.

– Subcase *viz* = **local**: similar

- Case (RED INDEX):

(RED INDEX)

$$\text{Core}(\mathbb{T}_f) \quad \text{NoQry}(\mathbb{T}_f)$$

$$(\mathbb{T}_f R)[e_{\text{index}} < e_{\text{size}}] \rightarrow (\text{index}_{\emptyset}(\mathbb{T}_f, e_{\text{index}}, e_{\text{size}})) R$$

The judgment $\Gamma \vdash^{\text{inst}} (\mathbb{T}_f R)[e_{\text{index}} < e_{\text{size}}] : Q$ must have been derived with

(MODEL INDEXED), which also implies $Q = Q^*[e_{\text{size}}]$ for some Q^* :

(MODEL INDEXED)

$$\frac{\Gamma \vdash^{pc} (\mathbb{T}_f R) : Q^* \quad \Gamma \vdash^{pc} e_{index} : \mathbf{mod}(e_{size}) ! \mathbf{rnd} \quad \text{NoQry}(\mathbb{T}_f R)}{\Gamma \vdash^{pc} (\mathbb{T}_f R)[e_{index} < e_{size}] : Q^*[e_{size}]}$$

The assumption $\Gamma \vdash^{pc} (\mathbb{T}_f R) : Q^*$ of the above rule must have been derived with (MODEL APPL):

(MODEL APPL)

$$\frac{\Gamma \vdash^{pc} \mathbb{T}_f : Q_f \quad \mathbf{fun}(\mathbb{T}_f) \quad \Gamma \vdash^{pc} R : Q_f \rightarrow Q^*}{\Gamma \vdash^{pc} \mathbb{T}_f R : Q^*}$$

By Corollary 5, we have $\Gamma \vdash^{pc} \text{index}_{\emptyset}(\mathbb{T}_f, e_{index}, e_{size}) : Q_f[e_{size}]$. By the inversion of typing of $\Gamma \vdash^{pc} e_{index} : \mathbf{mod}(e_{size}) ! \mathbf{rnd}$ we have $\Gamma \vdash^{\mathbf{static}} e_{size} : \mathbf{int} ! \mathbf{det}$, so by Lemma 98, $\Gamma \vdash^{pc} R : Q_f[e_{size}] \rightarrow Q^*[e_{size}]$. Since $\mathbf{fun}(\text{index}_{\emptyset}(\mathbb{T}_f, e_{index}, e_{size}))$ easily follows from $\mathbf{fun}(\mathbb{T}_f)$, we can derive $\Gamma \vdash^{pc} \text{index}_{\emptyset}(\mathbb{T}_f, e_{index}, e_{size}) R : Q^*[e_{size}]$ by (MODEL APPL).

- Case (RED INDEX INNER):

(RED INDEX INNER)

$$\frac{M_1 \rightarrow M'_1}{M_1[e_{index} < e_{size}] \rightarrow M'_1[e_{index} < e_{size}]}$$

Here $\Gamma \vdash^{pc} M_1[e_{index} < e_{size}] : Q$ must have been derived with (MODEL INDEXED):

(MODEL INDEXED)

$$\frac{\Gamma \vdash^{pc} M_1 : Q_1 \quad \Gamma \vdash^{pc} e_{index} : \mathbf{mod}(e_{size}) ! \mathbf{rnd} \quad \text{NoQry}(M_1)}{\Gamma \vdash^{pc} M_1[e_{index} < e_{size}] : Q_1[e_{size}]}$$

where $Q = Q_1[e_{size}]$. By induction hypothesis, we have $\Gamma \vdash^{pc} M'_1 : Q_1$. We can easily show that $\text{NoQry}(M_1)$ and $M_1 \rightarrow M'_1$ imply $\text{NoQry}(M'_1)$. Thus, by (MODEL INDEXED), we have $\Gamma \vdash^{pc} M_1[e_{index} < e_{size}] : Q_1[e_{size}]$.

- Case (RED MODEL):

(RED MODEL)

$$\frac{M \rightarrow M'}{(c \triangleright x : T \ell \text{ viz } M) :: \mathbb{T}_1 \rightarrow (c \triangleright x : T \ell \text{ viz } M') :: \mathbb{T}_1}$$

Here, $\Gamma \vdash^{pc} (c \triangleright x : T \ell \text{ viz } M) :: \mathbb{T}_1 : Q$ must have been derived with either (TABLE OUTPUT) or (TABLE LOCAL):

- Subcase (TABLE OUTPUT):

(TABLE OUTPUT)

$$\frac{\Gamma \vdash^{\ell \wedge pc} M : Q_c \quad \Gamma, x : Q_c \vdash^{pc} \mathbb{T}_1 : Q_1 \quad Q_c = Q'_c @ [(\text{ret} \triangleright y : T \ell' \text{ output})] \quad \text{names}(c.Q_c) \cap \text{names}(Q_1)}{\Gamma \vdash^{pc} (c \triangleright x : T \ell \text{ output } M) :: \mathbb{T}_1 : (c.Q_c) @ Q_1}$$

By induction hypothesis, $\Gamma \vdash^{\ell \wedge pc} M' : Q_c$, which immediately yields $\Gamma \vdash^{pc}$
 $(c \triangleright x : T \ell \text{ output } M') :: \mathbb{T}_1 : (c.Q_c) @ Q_1$ by (TABLE OUTPUT).

– Subcase (TABLE LOCAL):

(TABLE LOCAL)

$$\frac{\Gamma \vdash^{\ell \wedge pc} M : Q_c \quad \Gamma, x : Q_c \vdash^{pc} \mathbb{T}_1 : Q \quad Q_c = Q'_c @ [(\text{ret} \triangleright y : T \ell' \text{ output})]}{\Gamma \vdash^{pc} (\varepsilon \triangleright x : T \ell \text{ local } M) :: \mathbb{T}_1 : Q}$$

By induction hypothesis, $\Gamma \vdash^{\ell \wedge pc} M' : Q_c$, so we have $\Gamma \vdash^{pc} (\varepsilon \triangleright x : T \ell \text{ local } M') :: \mathbb{T}_1 : Q$ by (TABLE LOCAL).

• Case (RED APPL LOCAL): Similar to (RED APPL OUTPUT)

• Case (RED APPL INPUT):

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

$$\frac{}{(o \triangleright x : T \ell \text{ viz } (c \triangleright y : T' \ell' \text{ input } \varepsilon) :: \mathbb{T}_f (c = e) :: R) :: \mathbb{T}' \rightarrow (o \triangleright x : T \ell \text{ viz } \mathbb{T}_f \{e/y\} R) :: \mathbb{T}'}$$

– Subcase $\text{viz} = \text{output}$:

Here, $\Gamma \vdash^{\text{inst}} \mathbb{T} : Q$ must have been derived with:

(TABLE OUTPUT)

$$(1) \Gamma \vdash^{\ell} (c \triangleright y : T' \ell' \text{ input } \varepsilon) :: \mathbb{T}_f (c = e) :: R : Q_f$$

$$(2) \Gamma, x : Q_f \vdash^{\text{inst}} \mathbb{T}' : Q_1$$

$$(3) Q_f = Q'_f @ [(\text{ret} \triangleright y : T \ell'' \text{ output})]$$

$$(4) \text{names}(o.Q_f) \cap \text{names}(Q_1)$$

$$\frac{}{\Gamma \vdash^{\text{inst}} (o \triangleright x : T \ell \text{ output } ((c \triangleright y : T' \ell' \text{ input } \varepsilon) :: \mathbb{T}_f (c = e) :: R) :: \mathbb{T}' : (o.Q_f) @ Q_1}$$

Assumption (1) must have been derived with:

(MODEL APPL)

$$(5) \Gamma \vdash^{\ell} (c \triangleright y : T' \ell' \text{ input } \varepsilon) :: \mathbb{T}_f : (c \triangleright y : T' (\ell \wedge \ell') \text{ input}) :: Q^*$$

$$(6) \text{fun}((c \triangleright y : T' \ell' \text{ input } \varepsilon) :: \mathbb{T}_f)$$

$$(7) \Gamma \vdash^{\ell} (c = e) :: R : (c \triangleright y : T' (\ell \wedge \ell') \text{ input}) :: Q^* \rightarrow Q_f$$

$$\frac{}{\Gamma \vdash^{\ell} (c \triangleright y : T' \ell' \text{ input } \varepsilon) :: \mathbb{T}_f (c = e) :: R : Q_f}$$

Assumption (5) must have been derived with:

(TABLE INPUT)

$$\frac{\Gamma, y : \ell \wedge \ell' \quad T' \vdash^\ell \mathbb{T}_f : Q^* \quad c \notin \text{names}(Q^*)}{\Gamma \vdash^\ell (c \triangleright y : T' \ell' \text{ input } \varepsilon) :: \mathbb{T}_f : (c \triangleright y : T' (\ell \wedge \ell') \text{ input}) :: Q^*}$$

and (7) with:

(ARG INPUT)

$$\frac{\Gamma \vdash^{\ell \wedge \ell'} e : T \quad \Gamma \vdash^\ell R : Q^* \{e/y\} \rightarrow Q_f}{\Gamma \vdash^\ell ((c = e) :: R) : ((c \triangleright y : T' (\ell \wedge \ell') \text{ input}) :: Q^*) \rightarrow Q_f}$$

By Lemma 103, we have $\Gamma \vdash^\ell \mathbb{T}_f \{e/y\} : Q^* \{e/y\}$. As the **fun** predicate is obviously preserved by substitution, (MODEL APPL) yields $\Gamma \vdash^\ell \mathbb{T}_f \{e/y\} R : Q_f$. Hence, by (TABLE OUTPUT), we have $\Gamma \vdash^{\text{inst}} (o \triangleright x : T \ell \text{ output } \mathbb{T}_f \{e/y\} R) :: \mathbb{T}' : (o.Q_f) @ Q_1$, as required.

– Subcase $\text{viz} = \text{local}$: similar.

- Case (RED APPL RET):

(RED APPL RET)

$$\frac{}{(o \triangleright x : T \ell \text{ viz } [(\text{ret} \triangleright y : T' \ell' \text{ output } E)] []) :: \mathbb{T}' \rightarrow (o \triangleright x : T' \ell \wedge \ell' \text{ viz } E) :: \mathbb{T}'}$$

– Subcase $\text{viz} = \text{output}$:

The judgment $\Gamma \vdash^{\text{inst}} (o \triangleright x : T \ell \text{ output } [(\text{ret} \triangleright y : T' \ell' \text{ output } E)] []) :: \mathbb{T}' : Q$ must have been derived with (TABLE OUTPUT):

(TABLE OUTPUT)

$$\frac{\Gamma \vdash^\ell [(\text{ret} \triangleright y : T' \ell' \text{ output } E)] [] : Q_c \quad \Gamma, x : Q_c \vdash^{\text{inst}} \mathbb{T}' : Q_1 \quad Q_c = Q'_c @ [(\text{ret} \triangleright y : T' \ell'' \text{ output})] \quad \text{names}(o.Q_c) \cap \text{names}(Q_1) = \text{emptyset}}{\Gamma \vdash^{\text{inst}} (o \triangleright x : T \ell \text{ output } [(\text{ret} \triangleright y : T' \ell' \text{ output } E)] []) :: \mathbb{T}' : (o.Q_c) @ Q_1}$$

where $Q = (o.Q_c) @ Q_1$. The first assumption must have been derived with

(MODEL APPL):

(MODEL APPL)

$$\frac{\Gamma \vdash^\ell [(\text{ret} \triangleright y : T' \ell' \text{ output } E)] : [(\text{ret} \triangleright y : T' (\ell \wedge \ell') \text{ output})] \quad \text{fun}([(\text{ret} \triangleright y : T' (\ell \wedge \ell') \text{ output } E)]) \quad \Gamma \vdash^\ell [] : [(\text{ret} \triangleright y : T' (\ell \wedge \ell') \text{ output})] \rightarrow [(\text{ret} \triangleright y : T' (\ell \wedge \ell') \text{ output})]}{\Gamma \vdash^\ell [(\text{ret} \triangleright y : T' \ell' \text{ output } E)] [] : [(\text{ret} \triangleright y : T' (\ell \wedge \ell') \text{ output})]}$$

where $Q_c = [(\text{ret} \triangleright y : T' (\ell \wedge \ell') \text{ output})]$. By the last assumption of (TABLE OUTPUT), $T = T'$, so $Q_c = [(\text{ret} \triangleright y : T (\ell \wedge \ell') \text{ output})]$.

The first judgment above must have been derived with (TABLE CORE OUTPUT), and by inversion of typing, we have $\Gamma \vdash^{\ell \wedge \ell'} E : T$ and $\Gamma, y : \ell \wedge \ell' \vdash^{\ell} [] : []$, which implies $y \notin \text{dom}(\Gamma)$. By Lemma 108, we have $\Gamma, x : \ell \wedge \ell' \vdash^{\text{inst}} \mathbb{T}' : Q_1$. Thus, by (TABLE CORE OUTPUT), we have $\Gamma \vdash^{\text{inst}} (o \triangleright x : T (\ell \wedge \ell') \text{ output } E) :: \mathbb{T}' : (o \triangleright x : T (\ell \wedge \ell') \text{ output}) :: Q_1$. By the derived judgments, we can easily show that $\text{fv}(Q_1) \subseteq \text{dom}(\Gamma)$, which implies $x \notin \text{fv}(Q_1)$ and $y \notin \text{fv}(Q_1)$. Hence we have $(o \triangleright x : T (\ell \wedge \ell') \text{ output}) :: Q_1 =_{\alpha} (o \triangleright y : T (\ell \wedge \ell') \text{ output}) :: Q_1 = (o.Q_c)@Q_1$, and so $\Gamma \vdash^{\text{inst}} (o \triangleright x : T (\ell \wedge \ell') \text{ output } E) :: \mathbb{T}' : (o.Q_c)@Q_1$, as required.

– Subcase *viz* = **local**: similar.

- Case (RED TABLE RIGHT):

(RED TABLE RIGHT)

$$\frac{\mathbb{T}_1 \rightarrow \mathbb{T}'_1 \quad \text{Core}(\text{col})}{\text{col} :: \mathbb{T}_1 \rightarrow \text{col} :: \mathbb{T}'_1}$$

– Subcase: $\Gamma \vdash^{\text{inst}} \text{col} :: \mathbb{T}_1 : Q$ derived with (TABLE CORE OUTPUT):

(TABLE CORE OUTPUT)

$$\frac{\Gamma \vdash^{\ell} E : T \quad \Gamma, x : \ell \vdash^{\text{inst}} \mathbb{T}_1 : Q_1 \quad c \notin \text{names}(Q_1)}{\Gamma \vdash^{\text{inst}} (c \triangleright x : T \ell \text{ output } E) :: \mathbb{T}_1 : (c \triangleright x : T (\ell \wedge pc) \text{ output}) :: Q_1}$$

where $Q = (c \triangleright x : T (\ell \wedge pc) \text{ output}) :: Q_1$. By induction hypothesis, $\Gamma, x : \ell \vdash^{\text{inst}} \mathbb{T}'_1 : Q_1$, hence $\Gamma \vdash^{\text{inst}} \text{col} :: \mathbb{T}'_1 : Q$ follows immediately from (TABLE CORE OUTPUT).

– Subcases for (TABLE CORE LOCAL) and (TABLE INPUT) similar.

■

B.2 Proposition 2

To prove the progress property, we need the following lemma:

Lemma 111 *If $\Gamma \vdash^{pc} M[e_{\text{index}} < e_{\text{size}}] : Q[e_{\text{size}}]$, then there exists M' such that $M[e_{\text{index}} < e_{\text{size}}] \rightarrow M'$.*

Proof: By induction on the derivation of $\Gamma \vdash^{\ell \wedge pc} M[e_{index} < e_{size}] : Q[e_{size}]$:

We know that $\Gamma \vdash^{\ell \wedge pc} M[e_{index} < e_{size}] : Q[e_{size}]$ must have been derived with (MODEL INDEXED):

(MODEL INDEXED)

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

If $\Gamma \vdash^{pc} M : Q$ has also been derived with (MODEL INDEXED), then $M = M^*[e_1 < e_2]$ and $Q = Q^*[e_2]$ for some M^*, Q^* , and so by induction hypothesis, $M \rightarrow M^*$ for some M^* . Hence, by (RED INDEX INNER), we have $M[e_{index} < e_{size}] \rightarrow M^*[e_{index} < e_{size}]$.

If $\Gamma \vdash^{pc} M : Q$ has also been derived with (MODEL APPL), we have $M = \mathbb{T} R$ for some \mathbb{T} (satisfying $\mathbf{NoQry}(\mathbb{T})$ and $\mathbf{Core}(\mathbb{T})$) and R . Since index_{\emptyset} is a total function on \mathbf{Core} , \mathbf{NoQry} tables and indexed expressions, by (RED INDEX) we have $(\mathbb{T} R)[e_{index} < e_{size}] \rightarrow \text{index}_{\emptyset}(\mathbb{T}, e_{index}, e_{size}) R$.

■

Restatement of Proposition 2 [Progress]

- (1) If $\Gamma \vdash^{pc} \mathbb{T} : Q$ then either $\mathbf{Core}(\mathbb{T})$ or there is \mathbb{T}' such that $\mathbb{T} \rightarrow \mathbb{T}'$.
- (2) If $\Gamma \vdash^{pc} \mathbb{S} : \text{Sty}$ then either $\mathbf{Core}(\mathbb{S})$ or there is \mathbb{S}' such that $\mathbb{S} \rightarrow \mathbb{S}'$.

Proof: 1.) By induction on the derivation of $\Gamma \vdash^{pc} \mathbb{T} : Q$:

- Case (TABLE OUTPUT):

(TABLE OUTPUT)

$$\frac{\begin{array}{l} \Gamma \vdash^{\ell \wedge pc} M : Q_c \quad \Gamma, x : Q_c \vdash^{pc} \mathbb{T}' : Q \quad Q_c = Q'_c @ [(\text{ret} \triangleright y : T \ell' \mathbf{output})] \\ \text{names}(c.Q_c) \cap \text{names}(Q) = \emptyset \end{array}}{\Gamma \vdash^{pc} (c \triangleright x : T \ell' \mathbf{output} M) :: \mathbb{T}' : (c.Q_c) @ Q}$$

We need to split on derivation of $\Gamma \vdash^{\ell \wedge pc} M : Q_c$:

- Subcase (MODEL APPL):

(MODEL APPL)

$$\frac{\Gamma \vdash^{\ell \wedge pc} \mathbb{T}_f : Q_f \quad \mathbf{fun}(\mathbb{T}_f) \quad \Gamma \vdash^{\ell \wedge pc} R : Q_f \rightarrow Q_c}{\Gamma \vdash^{\ell \wedge pc} \mathbb{T}_f R : Q_c}$$

- * If $\Gamma \vdash^{\ell \wedge pc} \mathbb{T}_f : Q_f$ was derived with (TABLE CORE OUTPUT), then $\mathbb{T}_f = (d \triangleright y : T' \ell' \text{ output } E) :: \mathbb{T}'_f$ and $Q_f = (d \triangleright y : T' \ell' \text{ output}) :: Q'_f$.

We need to consider two cases:

- If $\Gamma \vdash^{\ell \wedge pc} R : Q_f \rightarrow Q_c$ was derived with (ARG OUTPUT), then $d \neq \text{ret}$, and so $(c \triangleright x : T \ell \text{ output } (d \triangleright y : T' \ell' \text{ output } E) :: \mathbb{T}'_f R) :: \mathbb{T}' \rightarrow (c \triangleright x : T \ell \wedge \ell' \text{ output } E) :: (c \triangleright x : T \ell \text{ output } \mathbb{T}'_f R) :: \mathbb{T}' \langle y/x.c \rangle$ by (RED APPL OUTPUT) (note that we can always rename y to make it sufficiently fresh to apply this rule).
- If $\Gamma \vdash^{\ell \wedge pc} R : Q_f \rightarrow Q_c$ was derived with (ARG RET), then $d = \text{ret}$ and $Q'_f = []$. From $\text{fun}((\text{ret} \triangleright y : T' \ell' \text{ output } E) :: \mathbb{T}'_f)$ we can deduce $\mathbb{T}'_f = []$. Hence, by (RED APPL RET), we have $(c \triangleright x : T \ell \text{ output } [(\text{ret} \triangleright y : T' \ell' \text{ output } E)] []) :: \mathbb{T}' \rightarrow (c \triangleright x : T' (\ell \wedge \ell') \text{ output } E) :: \mathbb{T}'$, as required.
- * If $\Gamma \vdash^{\ell \wedge pc} \mathbb{T}_f : Q_f$ was derived with (TABLE CORE LOCAL), then $\mathbb{T}_f = (d \triangleright y : T' \ell' \text{ local } E) :: \mathbb{T}'_f$ so (after renaming y if necessary) we have $(c \triangleright x : T \ell \text{ output } (d \triangleright y : T' \ell' \text{ local } E) :: \mathbb{T}'_f R) :: \mathbb{T}' \rightarrow (d \triangleright y : T' \ell \wedge \ell' \text{ local } E) :: (c \triangleright x : T \ell \text{ output } \mathbb{T}'_f R) :: \mathbb{T}'$
- * If $\Gamma \vdash^{\ell \wedge pc} \mathbb{T}_f : Q_f$ was derived with (TABLE INPUT), then $\mathbb{T}_f = (d \triangleright y : T' \ell' \text{ input } \varepsilon) :: \mathbb{T}'_f$ and $Q_f = (d \triangleright y : T' \ell' \text{ input}) :: Q'_f$. Hence, $\Gamma \vdash^{\ell \wedge pc} R : Q_f \rightarrow Q_c$ must have been derived with (ARG INPUT), which implies $R = (d = e) :: R'$. Thus, by (RED APPL INPUT), we have $(c \triangleright x : T \ell \text{ output } ((d \triangleright y : T' \ell' \text{ input } \varepsilon) :: \mathbb{T}'_f) (d = e) :: R') :: \mathbb{T}' \rightarrow (c \triangleright x : T \ell \text{ output } \mathbb{T}'_f \{e/y\} R') :: \mathbb{T}'$.

– Subcase (MODEL INDEXED):

(MODEL INDEXED)

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

By Lemma 111, there exists M'' such that $M'[e_{\text{index}} < e_{\text{size}}] \rightarrow M''$. Hence, by (RED MODEL), we have $(c \triangleright x : T \ell \text{ output } M'[e_{\text{index}} < e_{\text{size}}]) :: \mathbb{T}' \rightarrow (c \triangleright x : T \ell \text{ output } M'') :: \mathbb{T}'$.

- Case (TABLE LOCAL): similar.

■

B.3 Proposition 3

In order to prove termination of reduction, we need to define some metric on schemas and show that it is strictly decreasing under reduction. Let us define a metric m as follows:

Metric on schemas: $m(\mathbb{S})$

$$m((t = \mathbb{T}) :: \mathbb{S}) \triangleq m(\mathbb{T}) + m(\mathbb{S})$$

$$m(\square) \triangleq 0$$

$$m((c \triangleright x : T \ell \text{ viz } M) :: \mathbb{T}) \triangleq m(M) + m(\mathbb{T})$$

$$m(\mathbb{T} R) \triangleq n(\mathbb{T})$$

$$m(E) = 0$$

$$m(\varepsilon) = 0$$

$$m(M[e_1 < e_2]) = m(M) + 1$$

$$n((c \triangleright x : T \ell \text{ viz } M) :: \mathbb{T}) \triangleq n(\mathbb{T}) + 1$$

$$n(\square) \triangleq 0$$

Lemma 112 For any A , e_1 , e_2 , if $\text{Core}(\mathbb{T})$ and $\text{NoQry}(\mathbb{T})$, then $n(\text{index}_{\emptyset}(\mathbb{T}, e_1, e_2)) = n(\mathbb{T})$

Proof: This is an easy induction on the structure of \mathbb{T} . ■

Lemma 113 (1) If $M \rightarrow M'$, then $m(M') < m(M)$

(2) If $\mathbb{T} \rightarrow \mathbb{T}'$, then $m(\mathbb{T}') < m(\mathbb{T})$

(3) If $\mathbb{S} \rightarrow \mathbb{S}'$, then $m(\mathbb{S}') < m(\mathbb{S})$

Proof: 1.) By induction on the derivation of $M \rightarrow M'$:

- Case :

(RED INDEX INNER)

$$M' \rightarrow M''$$

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

By induction hypothesis, $m(M'') < m(M')$, so $m(M''[e_{\text{index}} < e_{\text{size}}]) = m(M'') + 1 < m(M') + 1 = m(M''[e_{\text{index}} < e_{\text{size}}])$.

- Case:

(RED INDEX)

$\text{Core}(\mathbb{T}_f) \quad \text{NoQry}(\mathbb{T}_f)$

$(\mathbb{T}_f R)[e_{\text{index}} < e_{\text{size}}] \rightarrow (\text{index}_{\emptyset}(\mathbb{T}_f, e_{\text{index}}, e_{\text{size}})) R$

We have $m((\mathbb{T}_f R)[e_{\text{index}} < e_{\text{size}}]) = 1 + n(\mathbb{T}_f)$ and $m((\text{index}_{\emptyset}(\mathbb{T}_f, e_{\text{index}}, e_{\text{size}})) R) = n((\text{index}_{\emptyset}(\mathbb{T}_f, e_{\text{index}}, e_{\text{size}})))$, so the result follows immediately by Lemma 112.

- Case:

(RED INDEX EXPR)

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

We have $m(E[e_{\text{index}} < e_{\text{size}}]) = 1$ and $m(E) = 0$ (but this model expression is not well-typed anyway).

2.) By induction on the derivation of $\mathbb{T} \rightarrow \mathbb{T}'$, with appeal to part 1:

- Case:

(RED APPL OUTPUT) (for $\text{Core}(\mathbb{T}_1)$)

$y \notin \text{fv}(\mathbb{T}'_1, R) \cup \{x\} \quad c \neq \text{ret}$

$(o \triangleright x : T \ell \text{ viz } ((c \triangleright y : T' \ell' \text{ output } E) :: \mathbb{T}_1) R) :: \mathbb{T}'_1 \rightarrow$
 $(o.c \triangleright y : T' \ell \wedge \ell' \text{ viz } E) :: (o \triangleright x : T \ell \text{ viz } \mathbb{T}_1 R) :: \mathbb{T}'_1 \langle y/x.c \rangle$

We have $m((o.c \triangleright y : T' \ell \wedge \ell' \text{ viz } E) :: (o \triangleright x : T \ell \text{ viz } \mathbb{T}_1 R) :: \mathbb{T}'_1 \langle y/x.c \rangle) = n(\mathbb{T}_1) + m(\mathbb{T}'_1) \leq 1 + n(\mathbb{T}_1) + m(\mathbb{T}'_1) = m((o \triangleright x : T \ell \text{ viz } ((c \triangleright y : T' \ell' \text{ output } E) :: \mathbb{T}_1) R) :: \mathbb{T}'_1)$, as required.

- Case:

(RED APPL LOCAL) (for $\text{Core}(\mathbb{T}_1)$)

$y \notin \text{fv}(\mathbb{T}'_1, R) \cup \{x\}$

$(o \triangleright x : T \ell \text{ viz } ((p \triangleright y : T' \ell' \text{ local } E) :: \mathbb{T}_1) R) :: \mathbb{T}'_1 \rightarrow$
 $(p \triangleright y : T' \ell \wedge \ell' \text{ local } E) :: (o \triangleright x : T \ell \text{ viz } \mathbb{T}_1 R) :: \mathbb{T}'_1$

We have $m((p \triangleright y : T' \ell \wedge \ell' \text{ local } E) :: (o \triangleright x : T \ell \text{ viz } \mathbb{T}_1 R) :: \mathbb{T}'_1) = n(\mathbb{T}_1) + m(\mathbb{T}'_1) \leq 1 + n(\mathbb{T}_1) + m(\mathbb{T}'_1) = m((o \triangleright x : T \ell \text{ viz } ((p \triangleright y : T' \ell' \text{ local } E) :: \mathbb{T}_1) R) :: \mathbb{T}'_1)$.

- Case:

(RED APPL INPUT) (for $\text{Core}(\mathbb{T}_1)$)

$(o \triangleright x : T \ell \text{ viz } (c \triangleright y : T' \ell' \text{ input } \varepsilon) :: \mathbb{T}_1 (c = e) :: R) :: \mathbb{T}'_1 \rightarrow$
 $(o \triangleright x : T \ell \text{ viz } \mathbb{T}_1 \{e/y\} R) :: \mathbb{T}'_1$

Here, $m((o \triangleright x : T \ell \text{ viz } \mathbb{T}_1 \{e/y\} R) :: \mathbb{T}'_1) = n(\mathbb{T}_1) + m(\mathbb{T}'_1) \leq 1 + n(\mathbb{T}_1) + m(\mathbb{T}'_1) = m((o \triangleright x : T \ell \text{ viz } (c \triangleright y : T' \ell' \mathbf{input} \ \varepsilon) :: \mathbb{T}_1 \ (c = e) :: R) :: \mathbb{T}'_1)$, as required.

- Case:

(RED APPL RET)

$$\frac{(o \triangleright x : T \ell \text{ viz } [(ret \triangleright y : T' \ell' \mathbf{output} \ E)] [] :: \mathbb{T}'_1 \rightarrow (o \triangleright x : T' \ell \wedge \ell' \text{ viz } E) :: \mathbb{T}'_1)}{}$$

We have $m((o \triangleright x : T' \ell \wedge \ell' \text{ viz } E) :: \mathbb{T}'_1) = m(\mathbb{T}'_1) \leq 1 + m(\mathbb{T}'_1) = m((o \triangleright x : T \ell \text{ viz } [(ret \triangleright y : T' \ell' \mathbf{output} \ E)] [] :: \mathbb{T}'_1)$, as required.

- Case:

(RED TABLE RIGHT)

$$\frac{\mathbb{T} \rightarrow \mathbb{T}'_1 \quad \text{Core}(\text{col})}{\text{col} :: \mathbb{T}_1 \rightarrow \text{col} :: \mathbb{T}'_1}$$

By induction hypothesis, $n(\mathbb{T}'_1) < n(\mathbb{T})$, so $n(\text{col} :: \mathbb{T}'_1) = n(\mathbb{T}'_1) < n(\mathbb{T}) = n(\text{col} :: \mathbb{T}_1)$.

- Case:

(RED MODEL)

$$\frac{M_1 \rightarrow M'_1}{(c \triangleright x : T \ell \text{ viz } M_1) :: \mathbb{T}_1 \rightarrow (c \triangleright x : T \ell \text{ viz } M'_1) :: \mathbb{T}_1}$$

By part 1.) we have $m(M'_1) < m(M_1)$, so $m((c \triangleright x : T \ell \text{ viz } M'_1) :: \mathbb{T}_1) = m(M'_1) + m(\mathbb{T}_1) < m(M_1) + m(\mathbb{T}_1) = m((c \triangleright x : T \ell \text{ viz } M_1) :: \mathbb{T}_1)$.

3.) By induction on the derivation of $\mathbb{S} \rightarrow \mathbb{S}'$, with appeal to part 2:

- Case:

(RED SCHEMA LEFT)

$$\frac{\mathbb{T}_1 \rightarrow \mathbb{T}'_1}{(t = \mathbb{T}_1) :: \mathbb{S}_1 \rightarrow (t = \mathbb{T}'_1) :: \mathbb{S}_1}$$

By part 3.) we have $m(\mathbb{T}'_1) < m(\mathbb{T}_1)$, so $m((t = \mathbb{T}'_1) :: \mathbb{S}_1) = m(\mathbb{T}'_1) + m(\mathbb{S}_1) < m(\mathbb{T}_1) + m(\mathbb{S}_1) = m((t = \mathbb{T}_1) :: \mathbb{S}_1)$

- Case:

(RED SCHEMA RIGHT)

$$\frac{\mathbb{S}_1 \rightarrow \mathbb{S}'_1 \quad \text{Core}(\mathbb{T}_1)}{(t = \mathbb{T}_1) :: \mathbb{S}_1 \rightarrow (t = \mathbb{T}_1) :: \mathbb{S}'_1}$$

By induction hypothesis we have $m(\mathbb{S}'_1) < m(\mathbb{S}_1)$, so $m((t = \mathbb{T}_1) :: \mathbb{S}_1) = m(\mathbb{T}_1) + m(\mathbb{S}_1) < m(\mathbb{T}_1) + m(\mathbb{S}'_1) = m((t = \mathbb{T}_1) :: \mathbb{S}'_1)$.

■

Restatement of Proposition 3 *[Termination] There does not exist an infinite chain of reductions $\mathbb{S}_1 \rightarrow \mathbb{S}_2 \rightarrow \dots$*

Proof: Suppose such a chain exists. The measure m on schemas is non-negative by definition, so for all $i \in \mathbb{N}$ we have $m(\mathbb{S}_i) \geq 0$. By Lemma 113, $m(\mathbb{S}_{i+1}) < m(\mathbb{S}_i)$, so because the measure is integer-valued by construction, $m(\mathbb{S}_{i+1}) \leq m(\mathbb{S}_i) - 1$. We can then show by induction that $m(\mathbb{S}_{i+1}) \leq m(\mathbb{S}_1) - i$, from which we get $m(\mathbb{S}_{m(\mathbb{S}_1)+1}) \leq -1$, which is a contradiction, since the measure is non-negative. ■

Appendix C

Proof of Tabular Output Database Conformance

This appendix includes detailed proofs of properties of the random and query semantics of Tabular.

C.1 Random Semantics

In this section, we prove Lemma 7, which states that for every valid trace, the random semantics returns a database of well-typed values of expressions in queries.

Restatement of Lemma 7 *If $\text{Core}(\mathbb{S})$ and $\emptyset \vdash \mathbb{S} : \text{Sty}$ and $(\delta_{in}, \rho_{sz}) \models \text{Sty}$ and $(\delta_{in}, \rho_{sz}) \vdash \mathbb{S} \Downarrow_w^s \delta_{qry}$, then for every $t \in \text{dom}(\mathbb{S})$ and $(c \triangleright x : T \ell \text{ viz } \mathbf{infer}.D[e_1, \dots, e_m].c_j(E')) \in \text{cols}(\mathbb{S}(t))$:*

- *If $\ell = \mathbf{static}$, then $\delta_{qry}(t)(c) = \mathbf{static}(V)$ and $\emptyset \vdash^{\mathbf{static}} V : \mathbf{real} ! \mathbf{rnd}$.*
- *If $\ell = \mathbf{inst}$, then $\delta_{qry}(t)(c) = \mathbf{inst}([V_0, \dots, V_{\rho_{sz}(t)-1}])$ and $\emptyset \vdash^{\mathbf{inst}} V_1 : \mathbf{real} ! \mathbf{rnd}$ for all $i \in 0..\rho_{sz} - 1$.*

To prove this lemma, we need the following auxiliary definitions and results. In the rest of this appendix, we write $\tau(T)$ for the type T with all variables in the domain of τ whose entries in τ are scalars at **static** level substituted with their values in τ .

We begin by defining several auxiliary conformance relations relating evaluation environments to types and typing environments: $(\tau, \rho_{sz}) \models_n^{\mathbf{rnd}-t} Q$ states that the table-level map τ with table size n conforms to the table type Q ; $\rho_{sz}; \delta; \tau \models_n^{\mathbf{rnd}} \Gamma$ states that the evaluation environment consisting of schema-level map δ and table-level map τ

conforms to the typing environment Γ , assuming that the currently evaluated table has n rows; $(\rho_{sz}; \delta; \tau) \models_n^{rnd-loc} (\Gamma; \Gamma')$ is similar, except that the typing environment is split in two parts Γ and Γ' , where Γ' contains local variables (that is, iterator variables in **for**-loops). The last judgment is needed to allow storing a single value for an **inst**-level iterator variable in the database during evaluation.

Table-level Random Database Conformance: $(\tau, \rho_{sz}) \models_n^{rnd-t} Q$

<hr/>	
(CONF SKIP RND)	(CONF LOCAL RND)
$\mathbf{qry}(T) \quad (\tau; \rho_{sz}) \models_n^{rnd-t} Q$	$(\tau; \rho_{sz}) \models_n^{rnd-t} Q$
$(\tau; \rho_{sz}) \models_n^{rnd-t} (c \triangleright x : T \ell \text{ viz}) :: Q$	$((c \mapsto \ell(V)) :: \tau; \rho_{sz}) \models_n^{rnd-t} Q$
<hr/>	
(CONF STATIC INOUT RND)	
$\text{viz} \in \{\mathbf{input}, \mathbf{output}\} \quad \emptyset \vdash^{\mathbf{static}} V : \rho_{sz}(T)$	
$(\tau; \rho_{sz}) \models_n^{rnd-t} Q$	
$((c \mapsto \mathbf{static}(V)) :: \tau; \rho_{sz}) \models_n^{rnd-t} (c \triangleright x : T \mathbf{static} \text{ viz}) :: Q$	
<hr/>	
(CONF INST INOUT RND)	
$\text{viz} \in \{\mathbf{input}, \mathbf{output}\} \quad \emptyset \vdash^{\mathbf{inst}} V_i : \rho_{sz}(T) \quad \forall i \in 0..n-1$	
$(\tau; \rho_{sz}) \models_n^{rnd-t} Q$	
$((c \mapsto \mathbf{inst}([V_1, \dots, V_n])) :: \tau; \rho_{sz}) \models_n^{rnd-t} (c \triangleright x : T \mathbf{inst} \text{ viz}) :: Q$	
<hr/>	
(CONF \square RND)	
$(\square; \rho_{sz}) \models_n^{rnd-t} \square$	
<hr/>	

Conformance of Random Database to Environment: $\rho_{sz}; \delta; \tau \models_n^{rnd} \Gamma$

<hr/>	
(RND VAR STATIC) (where $\neg \mathbf{qry}(T)$)	
$\emptyset \vdash^{\mathbf{static}} V : \rho_{sz}(\tau(T))$	
$(\rho_{sz}; \delta; \tau) \models_n^{rnd} \Gamma$	
$(\rho_{sz}; \delta; \tau, (x \mapsto \mathbf{static}(V))) \models_n^{rnd} \Gamma, x : \mathbf{static} T$	
<hr/>	
(RND VAR INST) (where $\neg \mathbf{qry}(T)$)	
$\emptyset \vdash^{\mathbf{inst}} V_i : \rho_{sz}(\tau(T)) \quad \forall i \in 0..n-1$	
$(\rho_{sz}; \delta; \tau) \models_n^{rnd} \Gamma$	
$(\rho_{sz}; \delta; \tau, (x \mapsto \mathbf{inst}([V_1, \dots, V_n]))) \models_n^{rnd} \Gamma, x : \mathbf{inst} T$	

(RND VAR QRY) (where $\mathbf{qry}(T)$)

$$\frac{(\rho_{sz}; \delta; \tau) \models_n^{rnd} \Gamma}{(\rho_{sz}; \delta; \tau) \models_n^{rnd} \Gamma, x :^\ell T}$$

(RND TABLE)

$$\frac{\begin{array}{l} \rho_{sz}(t) \in \mathbb{N} \\ (\tau_t, \rho_{sz}) \models_{\rho_{sz}(t)}^{rnd-t} Q \\ (\rho_{sz}; \delta; \tau) \models_n^{rnd} \Gamma \end{array}}{(\rho_{sz}; \delta, (t \mapsto \tau_t); \tau) \models_n^{rnd} \Gamma, t : Q} \quad \text{(RND EMPTY)} \quad \frac{}{(\rho_{sz}; []; []) \models_n^{rnd} \emptyset}$$

Conformance of Random Database to Local Environment: $(\rho_{sz}; \delta; \tau) \models_n^{rnd-loc} (\Gamma; \Gamma')$

(RND VAR LOCAL)

$$\frac{\begin{array}{l} \emptyset \vdash^{\mathbf{static}} V : \rho_{sz}(\tau(T)) \\ (\rho_{sz}; \delta; \tau) \models_n^{rnd} (\Gamma; \Gamma') \end{array}}{(\rho_{sz}; \delta; \tau, (x \mapsto \mathbf{static}(V))) \models_n^{rnd-loc} (\Gamma; \Gamma', x :^\ell T)}$$

(RND VAR LOCAL EMPTY)

$$\frac{(\rho_{sz}; \delta; \tau) \models_n^{rnd} \Gamma}{(\rho_{sz}; \delta; \tau) \models_n^{rnd-loc} (\Gamma; \emptyset)}$$

The below lemma states that evaluating random expressions with valid traces yields well-typed values.

Lemma 114 *If $\Gamma, \Gamma' \vdash^{pc} E : T$ and $(\rho_{sz}; \delta; \tau') \models_{\rho_{sz}(t)}^{rnd-loc} (\Gamma; \Gamma')$ and $\delta, \tau', i \vdash E \Downarrow_w^s V_i$ for some $i \in 0..\rho_{sz}(t) - 1$ then $\emptyset \vdash^{pc} V_i : \rho_{sz}(\tau'(T))$.*

Proof: By induction on the derivation of $\Gamma, \Gamma' \vdash^{pc} E : T$. The proof is similar to the proof of Lemma 127, with the fail cases removed. ■

Corollary 6 *If $\Gamma \vdash^{pc} E : T$ and $(\rho_{sz}; \delta; \tau') \models_{\rho_{sz}(t)}^{rnd} \Gamma$ and $\delta, \tau', i \vdash E \Downarrow_w^s V_i$ for some $i \in 0..\rho_{sz}(t) - 1$ then $\emptyset \vdash^{pc} V_i : \rho_{sz}(\tau'(T))$.*

The following lemma shows that the table-level database τ containing deterministic and random variables, obtained by evaluating a well-typed table with a valid trace, conforms to the type of the said table.

Lemma 115 *If $\text{Core}(\mathbb{T})$ and $\Gamma \vdash^{\mathbf{inst}} \mathbb{T} : Q$ and $\mathbf{table}(Q)$ and $\text{fv}(Q) = \emptyset$ and $(\delta_{in}(t), \rho_{sz}) \models_{\rho_{sz}(t)}$ Q and $t; \delta; \tau' \vdash \mathbb{T} \Downarrow_w^s \tau, \tau_{qry}$ and $(\rho_{sz}; \delta; \tau') \models_{\rho_{sz}(t)}^{rnd} \Gamma$ then $(\tau, \rho_{sz}) \models_{\rho_{sz}(t)}^{rnd-t} Q$*

Proof: By induction on the derivation of $t; \delta; \tau' \vdash \mathbb{T} \Downarrow_w^s \tau, \tau_{qry}$:

- Case (EVAL OUTPUT):

(EVAL OUTPUT) (where $\neg \mathbf{qry}(T)$)

$$\delta; \tau'; \delta_{in}(t)(c) \vdash E \Downarrow_{w_1}^{s_1} V$$

$$t; \delta; \tau', (x \mapsto \ell(V)) \vdash \mathbb{T}' \Downarrow_{w_2}^{s_2} \hat{\tau}, \tau_{qry}$$

$$t; \delta; \tau' \vdash (c \triangleright x : T \ell \text{ output } E) :: \mathbb{T}' \Downarrow_{w_1 w_2}^{s_1 @ s_2} [c \mapsto \ell(V)] @ \hat{\tau}, \tau_{qry}$$

Here, $\Gamma \vdash^{\mathbf{inst}} (c \triangleright x : T \ell \text{ output } E) :: \mathbb{T}' : Q$ must have been derived with (TABLE CORE OUTPUT), so $Q = (c \triangleright x : T \ell \text{ output}) :: Q'$. By the derivation of this typing judgment, we have $\Gamma, x : \ell T \vdash^{\mathbf{inst}} \mathbb{T}' : Q'$ and $\Gamma \vdash^\ell E : T$. By the assumption $\neg \mathbf{qry}(T)$, we know that $E \neq \mathbf{infer}.D[e_1, \dots, e_m].c_j(E')$.

- Subcase $\ell = \mathbf{static}$:

In this case, $\delta; \tau'; \delta_{in}(t)(c) \vdash E \Downarrow_{w_1}^{s_1} V$ must have been derived with (EVAL STATIC), and the assumption of that rule, $\delta; \tau'; \delta_{in}(t)(c) \vdash E \Downarrow_{w_1}^{s_1} V$ must have been derived with either (EVAL SAMPLE) or (EVAL COND):

- * Subcase (EVAL SAMPLE): Here, we have $\delta; \tau'; 0 \vdash E \Downarrow_{w_1}^{s_1} V$. By Corollary 6 and the fact that T is a top-level type of an **output** column, $\emptyset \vdash^{\mathbf{static}} V : \rho_{sz}(T)$.

By (RND VAR STATIC), $(\rho_{sz}; \delta; \tau', (x \mapsto \mathbf{static}(V))) \models_{\rho_{sz}(t)}^{rnd} \Gamma, x : \mathbf{static} T$.

By induction hypothesis, $(\hat{\tau}, \rho_{sz}) \models_{\rho_{sz}(t)}^{rnd-t} Q'$, so by (CONF STATIC IN-OUT RND), $((c \mapsto \ell(V)) :: \hat{\tau}, \rho_{sz}) \models_{\rho_{sz}(t)}^{rnd-t} (c \triangleright x : T \ell \text{ output}) :: Q'$,

- * Subcase (EVAL COND): Here, $\delta_{in}(t)(c) = \mathbf{static}(c')$ and $V = c'$, where $c' \in \mathbb{R}$. By the derivation of $(\delta_{in}(t), \rho_{sz}) \models_{\rho_{sz}(t)} Q$ we have $\emptyset \vdash^{\mathbf{static}} c' : \rho_{sz}(T)$. Hence, we have $(\rho_{sz}; \delta; \tau', (x \mapsto \mathbf{static}(V))) \models_{\rho_{sz}(t)}^{rnd} \Gamma, x : \mathbf{static} T$, again by (RND VAR STATIC), so by induction hypothesis, $(\hat{\tau}, \rho_{sz}) \models_{\rho_{sz}(t)}^{rnd-t} Q'$, and so $((c \mapsto \ell(V)) :: \hat{\tau}, \rho_{sz}) \models_{\rho_{sz}(t)}^{rnd-t} (c \triangleright x : T \ell \text{ output}) :: Q'$ by (CONF STATIC INOUT RND).

- Subcase $\ell = \mathbf{inst}$: Similar to $\ell = \mathbf{static}$.

- Case (EVAL LOCAL): Similar to (EVAL OUTPUT).
- Case (EVAL EMPTY): trivial.
- Case (EVAL INPUT):

(EVAL INPUT) (where $\neg \mathbf{qry}(T)$)

$$\frac{t; \delta; \tau', (x \mapsto \delta_{in}(t)(c)) \vdash \mathbb{T}' \Downarrow_w^s \hat{\tau}, \tau_{qry}}{t; \delta; \tau' \vdash (c \triangleright x : T \ell \text{ input } \varepsilon) :: \mathbb{T}' \Downarrow_w^s [c \mapsto \delta_{in}(t)(c)] @ \hat{\tau}, \tau_{qry}}$$

Here, $\Gamma \vdash^{\mathbf{inst}} (c \triangleright x : T \ell \text{ input } \varepsilon) :: \mathbb{T}' : Q$ must have been derived with (TABLE INPUT), so $Q = (c \triangleright x : T \ell \text{ input}) :: Q'$ and $\Gamma, x : ^\ell T \vdash^{\mathbf{inst}} \mathbb{T}' : Q$.

- Subcase $\ell = \mathbf{static}$: By the derivation of $(\delta_{in}(t), \rho_{sz}) \models_{\rho_{sz}(t)} Q$, we have $\delta_{in}(t)(c) = \mathbf{static}(V)$ and $\emptyset \vdash^{\mathbf{static}} V : \rho_{sz}(T)$. Hence, $(\rho_{sz}; \delta; \tau', (x \mapsto \mathbf{static}(V))) \models_{\rho_{sz}(t)}^{\mathbf{rnd}} \Gamma, x : \mathbf{static} T$, by (RND VAR STATIC). Then, by (CONF STATIC INOUT RND), $((c \mapsto \ell(V)) :: \hat{\tau}, \rho_{sz}) \models_{\rho_{sz}(t)}^{\mathbf{rnd}-t} (c \triangleright x : T \ell \text{ output}) :: Q'$,
- Subcase $\ell = \mathbf{inst}$: similar.

- Case (EVAL QUERY):

(EVAL QUERY)

$\mathbf{qry}(T)$

$E = \mathbf{infer}.D[e_1, \dots, e_m].c_j(E')$

$\delta; \tau'; \ell \langle t \rangle \vdash E' \Downarrow_{w_1}^{s_1} V$

$t; \delta; \tau' \vdash \mathbb{T} \Downarrow_{w_2}^{s_2} \hat{\tau}, \tau'_{qry}$

$t; \delta; \tau' \vdash (c \triangleright x : T \ell \text{ viz } E) :: \mathbb{T} \Downarrow_{w_1 w_2}^{s_1 @ s_2} \hat{\tau}, \tau'_{qry} @ [c \mapsto \ell(V)]$

- Subcase $\text{viz} = \mathbf{output}$:

In this case, $\Gamma \vdash^{\mathbf{inst}} (c \triangleright x : T \ell \text{ viz } E) :: \mathbb{T} : Q$ must have been derived with (TABLE CORE OUTPUT), so we have $Q = (c \triangleright x : T \ell \text{ output}) :: Q'$. By the derivation of the typing judgment, $\Gamma, x : ^\ell T \vdash^{\mathbf{inst}} \mathbb{T}' : Q'$ and $\Gamma \vdash^\ell E : T$.

- * Subcase $\ell = \mathbf{static}$:

By (RND VAR QRY), $(\rho_{sz}; \delta; \tau') \models_{\rho_{sz}(t)}^{\mathbf{rnd}} \Gamma, x : \mathbf{static} T$, so by induction hypothesis, $(\hat{\tau}, \rho_{sz}) \models_{\rho_{sz}(t)}^{\mathbf{rnd}-t} Q'$, and so $(\hat{\tau}, \rho_{sz}) \models_{\rho_{sz}(t)}^{\mathbf{rnd}-t} (c \triangleright x : T \ell \text{ output}) :: Q'$ by (CONF SKIP RND).

- * Subcase $\ell = \mathbf{inst}$: similar.

- Subcase $\text{viz} = \mathbf{local}$: similar

- Case (EVAL SKIP): Similar to (EVAL QUERY).

■

Lemma 116 *If $\text{Core}(\mathbb{T})$ and $\Gamma \vdash^{\text{inst}} \mathbb{T} : Q$ and $\text{table}(Q)$ and $\text{fv}(Q) = \emptyset$ and $(\delta_{in}(t), \rho_{sz}) \models_{\rho_{sz}(t)} Q$ and $t; \delta; \tau' \vdash \mathbb{T} \Downarrow_w^s \tau, \tau_{qry}$ and $(\rho_{sz}; \delta; \tau') \models_{\rho_{sz}(t)}^{rnd} \Gamma$ then the identifiers in τ and τ_{qry} are unique.*

Proof: The proof is a straightforward induction on the derivation of $(\delta_{in}(t), \rho_{sz}) \models_{\rho_{sz}(t)} Q$, using the fact that external column names are distinct in well-formed Q -types. ■

This lemma states that the database of values of expressions in queries stores a well-typed value for every expression in a query:

Lemma 117 *If $\text{Core}(\mathbb{T})$ and $\Gamma \vdash^{\text{inst}} \mathbb{T} : Q$ and $\text{table}(Q)$ and $\text{fv}(Q) = \emptyset$ and $(\delta_{in}(t), \rho_{sz}) \models_{\rho_{sz}(t)} Q$ and $t; \delta; \tau' \vdash \mathbb{T} \Downarrow_w^s \tau, \tau_{qry}$ and $(\rho_{sz}; \delta; \tau') \models_{\rho_{sz}(t)}^{rnd} \Gamma$ then for every $(d \triangleright x : T \ell \text{ viz } \text{infer}.D[e_1, \dots, e_m].c_j(E')) \in \text{cols}(\mathbb{T})$:*

- If $\ell = \text{static}$, then $\tau_{qry}(d) = \text{static}(V)$ and $\emptyset \vdash^{\text{static}} V : \text{real} ! \text{rnd}$.
- If $\ell = \text{inst}$, then $\tau_{qry}(d) = \text{inst}([V_0, \dots, V_{\rho_{sz}(t)-1}])$ and $\emptyset \vdash^{\text{inst}} V_i : \text{real} ! \text{rnd}$ for all $i \in 1..\rho_{sz}(t) - 1$.

Proof: By induction on the derivation of $t; \delta; \tau' \vdash \mathbb{T} \Downarrow_w^s \tau, \tau_{qry}$:

- Case (EVAL OUTPUT):

(EVAL OUTPUT) (where $\neg \text{qry}(T)$)

$\delta; \tau'; \delta_{in}(t)(c) \vdash E \Downarrow_{w_1}^{s_1} V$

$t; \delta; \tau', (x \mapsto \ell(V)) \vdash \mathbb{T}' \Downarrow_{w_2}^{s_2} \hat{\tau}, \tau_{qry}$

$t; \delta; \tau' \vdash (c \triangleright x : T \ell \text{ output } E) :: \mathbb{T}' \Downarrow_{w_1 w_2}^{s_1 @ s_2} [c \mapsto \ell(V)] @ \hat{\tau}, \tau_{qry}$

Here, $\Gamma \vdash^{\text{inst}} (c \triangleright x : T \ell \text{ output } E) :: \mathbb{T}' : Q$ must have been derived with (TABLE CORE OUTPUT), so $Q = (c \triangleright x : T \ell \text{ output}) :: Q'$. By the derivation of the typing judgment, we have $\Gamma, x : \ell T \vdash^{\text{inst}} \mathbb{T}' : Q'$ and $\Gamma \vdash^\ell E : T$. By the assumption $\neg \text{qry}(T)$, we know that $E \neq \text{infer}.D[e_1, \dots, e_m].c_j(E')$.

- Subcase $\ell = \text{static}$:

In this case, $\delta; \tau'; \delta_{in}(t)(c) \vdash E \Downarrow_{w_1}^{s_1} V$ must have been derived with (EVAL STATIC), and the assumption of that rule, $\delta; \tau'; \delta_{in}(t)(c) \vdash E \Downarrow_{w_1}^{s_1} V$ must have been derived with either (EVAL SAMPLE) or (EVAL COND):

- * Subcase (EVAL SAMPLE): Here, we have $\delta; \tau'; 0 \vdash E \Downarrow_{w_1}^{s_1} V$. By Corollary 6 and the fact that T is a top-level type of an **output** column,

$\emptyset \vdash^{\mathbf{static}} V : \rho_{sz}(T)$. Hence, by (RND VAR STATIC), $(\rho_{sz}; \delta; \tau', (x \mapsto \mathbf{static}(V))) \models_{\rho_{sz}(t)}^{rnd} \Gamma, x :^{\mathbf{static}} T$. Therefore, the result follows by induction hypothesis.

* Subcase (EVAL COND): Here, $\delta_{in}(t)(c) = \mathbf{static}(c')$ and $V = c'$, where $c' \in \mathbb{R}$. By the derivation of $(\delta_{in}(t), \rho_{sz}) \models_{\rho_{sz}(t)} Q$ we have $\emptyset \vdash^{\mathbf{static}} c' : \rho_{sz}(T)$. Hence, we have $(\rho_{sz}; \delta; \tau', (x \mapsto \mathbf{static}(V))) \models_{\rho_{sz}(t)}^{rnd} \Gamma, x :^{\mathbf{static}} T$, again by (RND VAR STATIC), so the result follows by the induction hypothesis.

– Subcase $\ell = \mathbf{inst}$: Similar to $\ell = \mathbf{static}$, repeating the reasoning for every index.

• Case (EVAL LOCAL): Similar to (EVAL OUTPUT).

• Case (EVAL EMPTY): trivial.

• Case (EVAL INPUT):

(EVAL INPUT) (where $\neg \mathbf{qry}(T)$)

$$\frac{t; \delta; \tau', (x \mapsto \delta_{in}(t)(c)) \vdash \mathbb{T}' \Downarrow_w^s \hat{\tau}, \tau_{qry}}{t; \delta; \tau' \vdash (c \triangleright x : T \ell \mathbf{input} \varepsilon) :: \mathbb{T}' \Downarrow_w^s [c \mapsto \delta_{in}(t)(c)] @ \hat{\tau}, \tau_{qry}}$$

Here, $\Gamma \vdash^{\mathbf{inst}} (c \triangleright x : T \ell \mathbf{input} \varepsilon) :: \mathbb{T}' : Q$ must have been derived with (TABLE INPUT), so $Q = (c \triangleright x : T \ell \mathbf{input}) :: Q'$ and $\Gamma, x :^\ell T \vdash^{\mathbf{inst}} \mathbb{T}' : Q$.

– Subcase $\ell = \mathbf{static}$: By the derivation of $(\delta_{in}(t), \rho_{sz}) \models_{\rho_{sz}(t)} Q$, we have $\delta_{in}(t)(c) = \mathbf{static}(V)$ and $\emptyset \vdash^{\mathbf{static}} V : \rho_{sz}(T)$. Hence, $(\rho_{sz}; \delta; \tau', (x \mapsto \mathbf{static}(V))) \models_{\rho_{sz}(t)}^{rnd} \Gamma, x :^{\mathbf{static}} T$, by (RND VAR STATIC) and the desired result follows by the induction hypothesis.

– Subcase $\ell = \mathbf{inst}$: similar.

• Case (EVAL QUERY):

(EVAL QUERY)

$\mathbf{qry}(T)$

$E = \mathbf{infer}.D[e_1, \dots, e_m].c_j(E')$

$t; \delta; \tau'; \ell \langle t \rangle \vdash E' \Downarrow_{w_1}^{s_1} V$

$t; \delta; \tau' \vdash \mathbb{T} \Downarrow_{w_2}^{s_2} \hat{\tau}, \tau'_{qry}$

$$\frac{t; \delta; \tau' \vdash \mathbb{T} \Downarrow_{w_2}^{s_2} \hat{\tau}, \tau'_{qry}}{t; \delta; \tau' \vdash (c \triangleright x : T \ell \mathbf{viz} E) :: \mathbb{T} \Downarrow_{w_1 w_2}^{s_1 @ s_2} \hat{\tau}, \tau'_{qry} @ [c \mapsto \ell(V)]}$$

– Subcase $viz = \mathbf{output}$:

In this case, $\Gamma \vdash^{\mathbf{inst}} (c \triangleright x : T \ell \text{ viz } E) :: \mathbb{T} : Q$ must have been derived with (TABLE CORE OUTPUT), so we have $Q = (c \triangleright x : T \ell \mathbf{output}) :: Q'$. By the derivation of the typing judgment, $\Gamma, x :^\ell T \vdash^{\mathbf{inst}} \mathbb{T}' : Q'$ and $\Gamma \vdash^\ell E : T$.

Moreover, $\Gamma \vdash^\ell \mathbf{infer}.D[e_1, \dots, e_m].c_j(E') : T$ must have been derived with (INFER), so $\Gamma \vdash^\ell E' : \mathbf{real} ! \mathbf{rnd}$ (using the assumption that only continuous distributions are allowed).

* Subcase $\ell = \mathbf{static}$:

Here, $\delta; \tau'; \ell \langle t \rangle \vdash E' \Downarrow_{w_1}^{s_1} V$ must have been derived with (EVAL STATIC), and its assumption, $\delta; \tau'; 0; ? \vdash E' \Downarrow_{w_1}^{s_1} V$, with (EVAL SAMPLE). Hence, we have $\delta; \tau'; 0 \vdash E' \Downarrow_{w_1}^{s_1} V$. By Corollary 6, $\emptyset \vdash^{\mathbf{static}} V : \mathbf{real} ! \mathbf{rnd}$.

Hence, if $c = d$, then the result holds immediately (note that by Lemma 116, there is no entry for d in τ'_{qry}).

If $c \neq d$, then, by (RND VAR QRY), $(\rho_{sz}; \delta; \tau') \models_{\rho_{sz}(t)}^{rnd} \Gamma, x :^{\mathbf{static}} T$, so the conclusion follows by the induction hypothesis.

* Subcase $\ell = \mathbf{inst}$: similar.

– Subcase $viz = \mathbf{local}$: similar

• Case (EVAL SKIP):

(EVAL SKIP)

$\mathbf{qry}(T)$

$E \neq \mathbf{infer}.D[e_1, \dots, e_m].c_j(E')$

$t; \delta; \tau' \vdash \mathbb{T}' \Downarrow_w^s \tau, \tau_{qry}$

$t; \delta; \tau' \vdash (c \triangleright x : T \ell \text{ viz } E) :: \mathbb{T} \Downarrow_w^s \tau, \tau_{qry}$

– Subcase $viz = \mathbf{output}$:

In this case, $\Gamma \vdash^{\mathbf{inst}} (c \triangleright x : T \ell \text{ viz } E) :: \mathbb{T}' : Q$ must have been derived with (TABLE CORE OUTPUT), so $Q = (c \triangleright x : T \ell \text{ viz}) :: Q'$ and $\Gamma, x :^\ell T \vdash^{\mathbf{inst}} \mathbb{T}' : Q'$. By (RND VAR QRY), $(\rho_{sz}; \delta; \tau') \models_{\rho_{sz}(t)}^{rnd} \Gamma, x :^{\mathbf{static}} T$, so the desired result follows by the induction hypothesis.

■

We can use the above results to show that the full output database returned by the random semantics contains well-typed values for all expressions in queries.

Lemma 118 *If $\text{Core}(\mathbb{S})$ and $\Gamma \vdash \mathbb{S} : \text{Sty}$ and $(\delta_{in}, \rho_{sz}) \models \text{Sty}$ and Γ only contains entries of the form $(t : Q')$ and $(\delta, \rho_{sz}) \vdash \mathbb{S} \Downarrow_w^s \delta_{qry}$ and $(\rho_{sz}; \delta; []) \models_n^{rnd} \Gamma$ then for every $t \in \text{dom}(\mathbb{S})$ and $(c \triangleright x : T \ell \text{ viz } \text{infer}.D[e_1, \dots, e_m].c_j(E')) \in \text{cols}(\mathbb{S}(t))$:*

- If $\ell = \text{static}$, then $\delta_{qry}(t)(c) = \text{static}(V)$ and $\emptyset \vdash^{\text{static}} V : \text{real} ! \text{rnd}$.
- If $\ell = \text{inst}$, then $\delta_{qry}(t)(c) = \text{inst}([V_0, \dots, V_{\rho_{sz}(t)-1}])$ and $\emptyset \vdash^{\text{inst}} V : \text{real} ! \text{rnd}$.

Proof: By induction on the derivation of $(\delta_{in}, \rho_{sz}) \vdash \mathbb{S} \Downarrow_w^s \delta_{qry}$:

- Case (QUERY SCHEMA TABLE):

(EVAL SCHEMA TABLE)

$$\frac{t; \delta; \emptyset \vdash \mathbb{T} \Downarrow_{w_1}^{s_1} \tau_t, \tau_{tq} \quad (\delta, (t' \rightarrow \tau_t); \rho_{sz}) \vdash \mathbb{S}' \Downarrow_{w_2}^{s_2} \delta_{qry}}{(\delta, \rho_{sz}) \vdash (t' = \mathbb{T}) :: \mathbb{S}' \Downarrow_{w_1 w_2}^{s_1 @ s_2} [t' \rightarrow \tau_{tq}] @ \delta_{qry}}$$

Here, $\Gamma \vdash \mathbb{S} : \text{Sty}$ must have been derived with (SCHEMA TABLE), so we have $\Gamma \vdash^{\text{inst}} \mathbb{T} : Q$ and $\Gamma, t' : Q \vdash \mathbb{S}' : \text{Sty}'$, where $\text{Sty} = (t' : Q) :: \text{Sty}'$.

By the derivation of $(\delta_{in}, \rho_{sz}) \models \text{Sty}$, we have $(\delta_{in}(t), \rho_{sz}) \models_{\rho_{sz}(t)} Q$ and $(\delta_{in}, \rho_{sz}) \models \text{Sty}'$.

By Lemma 115, $(\tau_t, \rho_{sz}) \models_{\rho_{sz}(t)}^{rnd-t} Q$.

Meanwhile, by (RND TABLE) we have $(\rho_{sz}; \delta, (t' \rightarrow \tau_t); []) \models_n^{rnd} \Gamma, t' : Q$.

If $t = t'$, then the desired result holds by Lemma 117. Otherwise, it holds by induction hypothesis.

- Case (QUERY SCHEMA EMPTY): trivial.

■

Restatement of Lemma 7 *If $\text{Core}(\mathbb{S})$ and $\emptyset \vdash \mathbb{S} : \text{Sty}$ and $(\delta_{in}, \rho_{sz}) \models \text{Sty}$ and $(\delta_{in}, \rho_{sz}) \vdash \mathbb{S} \Downarrow_w^s \delta_{qry}$, then for every $t \in \text{dom}(\mathbb{S})$ and $(c \triangleright x : T \ell \text{ viz } \text{infer}.D[e_1, \dots, e_m].c_j(E')) \in \text{cols}(\mathbb{S}(t))$:*

- If $\ell = \text{static}$, then $\delta_{qry}(t)(c) = \text{static}(V)$ and $\emptyset \vdash^{\text{static}} V : \text{real} ! \text{rnd}$.
- If $\ell = \text{inst}$, then $\delta_{qry}(t)(c) = \text{inst}([V_0, \dots, V_{\rho_{sz}(t)-1}])$ and $\emptyset \vdash^{\text{inst}} V : \text{real} ! \text{rnd}$.

Proof: Corollary of Lemma 118.

■

C.2 Preservation Result for the Query Semantics

In this section, we prove that every output database returned by the query semantics of Tabular conforms to the schema type:

Restatement of Lemma 14 *If $\text{Core}(\mathbb{S})$ and $\emptyset \vdash \mathbb{S} : \text{Sty}$ and $(\delta_{in}; \rho_{sz}) \models \text{Sty}$ and $\delta_{in}; [] \vdash_{\sigma} \mathbb{S} \Downarrow \delta_{out}$, then $(\delta_{out}; \rho_{sz}) \models^{out} \text{Sty}$.*

Like in the random semantics, we need to introduce some auxiliary conformance relations. The judgment $(\rho_{sz}; \delta; \tau) \models_n^{int} \Gamma$ says that the evaluation environment consisting of δ and τ conforms to the typing environment Γ if the size of the currently evaluated table is n . The judgment $(\rho_{sz}; \delta; \tau) \models_n^{loc} (\Gamma; \Gamma')$ is similar, except that it is used when evaluating expressions inside columns—the environment Γ contains types of local variables (iterators in **for**-loops) and the judgment allows an **inst**-level iterator to have a single static variable in the evaluation environment.

Below, $\tau(T)$ denotes the type T with each variable x substituted by the scalar s if $\tau = \tau_1 @ [(x \mapsto \mathbf{static}(s))] @ \tau_2$.

Conformance of an Intermediate Database: $(\rho_{sz}; \delta; \tau) \models_n^{int} \Gamma$

(CONF VAR STATIC) (where $\neg \mathbf{rnd}(T)$)

$\emptyset \vdash^{\mathbf{static}} G : \rho_{sz}(\tau(T))$

$(\rho_{sz}; \delta; \tau; \eta) \models_n^{int} \Gamma \quad \mathbf{det}(T) \Rightarrow \text{value}(G)$

$(\rho_{sz}; \delta; \tau, (x \mapsto \mathbf{static}(G)); \eta) \models_n^{int} \Gamma, x : \mathbf{static} T$

(CONF VAR INST) (where $\neg \mathbf{rnd}(T)$)

$\emptyset \vdash^{\mathbf{inst}} G_i : \rho_{sz}(\tau(T)) \quad \forall i \in 0..n-1$

$(\rho_{sz}; \delta; \tau; \eta) \models_n^{int} \Gamma \quad \mathbf{det}(T) \Rightarrow \text{value}(G_i) \quad \forall i \in 0..n-1$

$(\rho_{sz}; \delta; \tau, (x \mapsto \mathbf{inst}([G_1, \dots, G_n])); \eta) \models_n^{int} \Gamma, x : \mathbf{inst} T$

(CONF VAR RND) (where $\mathbf{rnd}(T)$)

$(\rho_{sz}; \delta; \tau; \eta) \models_n^{int} \Gamma$

$(\rho_{sz}; \delta; \tau; \eta) \models_n^{int} \Gamma, x : {}^\ell T$

(CONF TABLE)

$\rho_{sz}(t) \in \mathbb{N}$

$(\tau_t, \rho_{sz}) \models_{\rho_{sz}(t)}^{out} Q$

$(\rho_{sz}; \delta; \tau; \eta) \models_n^{int} \Gamma$

$(\rho_{sz}; \delta, (t \mapsto \tau_t); \tau; \eta) \models_n^{int} \Gamma, t : Q$

(CONF EMPTY)

$(\rho_{sz}; []; \eta) \models_n^{int} \emptyset$

Conformance of an Intermediate Database to Local Environment: $(\rho_{sz}; \delta; \tau) \models_n^{loc} (\Gamma; \Gamma')$

(CONF VAR LOCAL)

$\emptyset \vdash^\ell s : \rho_{sz}(\tau(T))$

$(\rho_{sz}; \delta; \tau; \eta) \models_n^{loc} (\Gamma; \Gamma')$

$(\rho_{sz}; \delta; \tau, (x \mapsto \mathbf{static}(s)); \eta) \models_n^{loc} (\Gamma; \Gamma', x :^\ell T)$

(CONF VAR LOCAL EMPTY)

$(\rho_{sz}; \delta; \tau; \eta) \models_n^{int} \Gamma$

$(\rho_{sz}; \delta; \tau; \eta) \models_n^{loc} (\Gamma; \emptyset)$

We need some auxiliary lemmas:

Lemma 119 (1) If $(\rho_{sz}, \delta, \tau', \eta) \models_n^{int} \Gamma', x : \mathbf{static} T, \Gamma''$, and $\neg \mathbf{rnd}(T)$ then $\tau' = \tau'_1 @ [(x \mapsto \mathbf{static}(G))] @ \tau'_2$ and $\emptyset \vdash \mathbf{static} G : \rho_{sz}(\tau'(T))$ and $\text{value}(G)$ if $\det(T)$.

(2) If $(\rho_{sz}, \delta, \tau', \eta) \models_n^{int} \Gamma', x : \mathbf{inst} T, \Gamma''$ and $\neg \mathbf{rnd}(T)$ then $\tau' = \tau'_1 @ [(x \mapsto \mathbf{inst}([G_0, \dots, G_{n-1}]))] @ \tau'_2$ and $\emptyset \vdash \mathbf{inst} G_i : \rho_{sz}(\tau'(T))$ for all $i \in 0..n-1$, where $\text{value}(G_i)$ for all $i \in 0..n-1$ if $\det(T)$.

Proof:

(1) By induction on the derivation of $(\rho_{sz}, \delta, \tau', \eta) \models_n^{int} \Gamma', x : \mathbf{static} T, \Gamma''$.

(2) By induction on the derivation of $(\rho_{sz}, \delta, \tau', \eta) \models_n^{int} \Gamma', x : \mathbf{inst} T, \Gamma''$.

■

Lemma 120 (1) If $(\rho_{sz}; \delta; \tau'; \eta) \models_n^{loc} (\Gamma'_1, x : \mathbf{static} T, \Gamma''_1; \Gamma_2)$ and $\neg \mathbf{rnd}(T)$ then $\tau' = \tau'_1 @ [(x \mapsto \mathbf{static}(G))] @ \tau'_2$ and $\emptyset \vdash \mathbf{static} G : \rho_{sz}(\tau'(T))$ and $\text{value}(G)$ if $\det(T)$.

(2) If $(\rho_{sz}; \delta; \tau'; \eta) \models_n^{loc} (\Gamma'_1, x : \mathbf{inst} T, \Gamma''_1; \Gamma_2)$ and $\neg \mathbf{rnd}(T)$ then $\tau' = \tau'_1 @ [(x \mapsto \mathbf{inst}([G_0, \dots, G_{n-1}]))] @ \tau'_2$ and $\emptyset \vdash \mathbf{inst} G_i : \rho_{sz}(\tau'(T))$ for all $i \in 0..n-1$ $\text{value}(G_i)$ for all $i \in 0..n-1$ if $\det(T)$.

(3) If $(\rho_{sz}; \delta; \tau'; \eta) \models_n^{loc} (\Gamma_1; \Gamma'_2, x :^\ell T, \Gamma''_2)$, and $\neg \mathbf{rnd}(T)$ then $\tau' = \tau'_1 @ [(x \mapsto \mathbf{static}(s))] @ \tau'_2$ and $\emptyset \vdash^\ell s : \rho_{sz}(\tau'(T))$.

Proof:

- (1) By induction on the derivation of $(\rho_{sz}; \delta; \tau'; \eta) \models_n^{loc} (\Gamma'_1, x : \mathbf{static} T, \Gamma''_1; \Gamma_2)$, with appeal to Lemma 119.
- (2) By induction on the derivation of $(\rho_{sz}; \delta; \tau'; \eta) \models_n^{loc} (\Gamma'_1, x : \mathbf{inst} T, \Gamma''_1; \Gamma_2)$, with appeal to Lemma 119.
- (3) By induction on the derivation of $(\rho_{sz}; \delta; \tau'; \eta) \models_n^{int} (\Gamma_1; \Gamma'_2, x : {}^\ell T, \Gamma''_2)$

■

Lemma 121 *If $(\rho_{sz}, \delta, \tau', \eta) \models_n^{int} \Gamma', t : Q, \Gamma''$, then $\delta = \delta_1 @ [(t \mapsto \tau_t)] @ \delta_2$ and $\rho_{sz}(t) \in \mathbb{N}$ and $(\tau_t, \rho_{sz}) \models_{\rho_{sz}(t)}^{out} Q$.*

Proof: By induction on the derivation of $(\rho_{sz}, \delta, \tau', \eta) \vdash \Gamma', t : Q, \Gamma''$. The proof is straightforward, so details are omitted. ■

Lemma 122 *If $(\rho_{sz}, \delta, \tau', \eta) \models_n^{loc} (\Gamma; \Gamma')$ and $\Gamma, \Gamma' = \Gamma_1, t : Q, \Gamma_2$, then $\delta = \delta_1 @ [(t \mapsto \tau_t)] @ \delta_2$ and $\rho_{sz}(t) \in \mathbb{N}$ and $(\tau_t, \rho_{sz}) \models_{\rho_{sz}(t)}^{out} Q$.*

Proof: By induction on the derivation of $(\rho_{sz}, \delta, \tau', \eta) \models_n^{loc} (\Gamma; \Gamma')$, with appeal to Lemma 121. ■

Lemma 123 (1) *If $(\tau, \rho_{sz}) \models_n^{out} Q_1 @ [(c \triangleright x : T \mathbf{static} viz)] @ Q_2$ and $\neg \mathbf{rnd}(T)$ then $\tau = \tau_1 @ [(c \mapsto \mathbf{static}(G))] @ \tau_2$ and $\emptyset \vdash^{\mathbf{static}} G : \rho_{sz}(T)$. and $\mathbf{value}(G)$ if $\mathbf{det}(T)$.*

(2) *If $(\tau, \rho_{sz}) \models_n^{out} Q_1 @ [(c \triangleright x : T \mathbf{inst} viz)] @ Q_2$ and $\neg \mathbf{rnd}(T)$ then $\tau = \tau_1 @ [(c \mapsto \mathbf{inst}([G_0, \dots, G_{n-1}]))] @ \tau_2$ and $\emptyset \vdash^{\mathbf{inst}} G_i : \rho_{sz}(T)$ for all $i \in 0..n-1$ and $\mathbf{value}(G_i)$ for all $i \in 0..n-1$ if $\mathbf{det}(T)$.*

Proof: By induction on the derivation of $(\tau, \rho_{sz}) \models_n^{out} Q_1 @ [(c \triangleright x : T \mathbf{static} viz)] @ Q_2$, and $(\tau, \rho_{sz}) \models_n^{out} Q_1 @ [(c \triangleright x : T \mathbf{inst} viz)] @ Q_2$. Again, we elide the details. ■

Lemma 124 *If $\Gamma, \Gamma' \vdash^{\mathbf{static}} e : \mathbf{int} ! \mathbf{det}$ and $(\rho_{sz}; \delta; \tau'; \eta) \models_n^{loc} (\Gamma; \Gamma')$ and $\delta; \tau'; \eta; i \vdash e \Downarrow G$, then $\rho_{sz}(\tau'(e)) = G$ and $\mathbf{value}(G)$.*

Proof: By case analysis:

- If $\delta, \tau', \eta, i \vdash e \Downarrow c$ was derived with (QUERY CONST), then $e = G$, so the result follows trivially.
- If $\delta, \tau', \eta, i \vdash e \Downarrow G$ was derived with (QUERY VAR STATIC), then $e = x$ and $\tau' = \tau'_1 @ [(x \mapsto \mathbf{static}(G))] @ \tau'_2$, so $\rho_{sz}(\tau'(x)) = \rho_{sz}(G) = G$. Since $\Gamma \vdash^{\mathbf{static}} e : \mathbf{int} ! \mathbf{det}$ must have been derived with (INDEX VAR), followed by a finite number (possibly 0) of applications of (SUBSUM), we have $\Gamma, \Gamma' = \Gamma_1, x :^{\mathbf{static}} U, \Gamma_2$, where $\Gamma \vdash U <: \mathbf{int} ! \mathbf{det}$, which also implies $U = \mathbf{int} ! \mathbf{det}$. Hence, by Lemma 120 (part 1 or 3, depending on whether $x \in \text{dom}(\Gamma)$ or $x \in \text{dom}(\Gamma')$), we have $\text{value}(G)$.
- If $\delta, \tau', \eta, i \vdash e \Downarrow G$ was derived with (QUERY VAR INST), then $e = x$ and $\tau' = \tau'_1 @ [(c \mapsto \mathbf{inst}([G_0, \dots, G_{n-1}]))] @ \tau'_2$. Since $\Gamma \vdash^{\mathbf{static}} e : \mathbf{int} ! \mathbf{det}$ must have been derived with (INDEX VAR), followed by a finite number (possibly 0) of applications of (SUBSUM), we have $\Gamma, \Gamma' = \Gamma_1, x :^{\mathbf{static}} U, \Gamma_2$, where $\Gamma \vdash U <: \mathbf{int} ! \mathbf{det}$, which implies $U = \mathbf{int} ! \mathbf{det}$.

If $x \in \text{dom}(\Gamma)$, then by Lemma 120 (part 2) and the uniqueness of identifiers in τ' , we have $\tau'(x) = \mathbf{static}(G)$. If $x \in \text{dom}(\Gamma')$, then by Lemma 120 (part 3) and the uniqueness of identifiers in τ' , we also have $\tau'(x) = \mathbf{static}(G)$, so in either case we arrive at a contradiction. Hence, this case is not possible.

- If $\delta, \tau', \eta, i \vdash e \Downarrow n$ was derived with (QUERY SIZEOF), then $e = \mathbf{sizeof}(t)$ and $\rho_{sz}(t) = n$, so $\rho_{sz}(\tau'(\mathbf{sizeof}(t))) = \rho_{sz}(\mathbf{sizeof}(t)) = \rho_{sz}(t) = n$.

■

Lemma 125 *If $\Gamma \vdash^{\mathbf{static}} e : T$ and $(\rho_{sz}; \delta; \tau'; \eta) \models_{\rho_{sz}(t)}^{\text{int}} \Gamma$ and $t; \delta; \tau'; 0 \vdash e_k \Downarrow s_k$ for all $k \in 1..m$ and e is not of the form $\mathbf{sizeof}(t')$ and $\{x_1, \dots, x_m\} \cap (\bigcup_k \text{fv}(e_k)) = \emptyset$ and $\text{fv}(e\{e_1/x_1\} \dots \{e_m/x_m\}) = \emptyset$, then $\rho_{sz}(e\{e_1/x_1\} \dots \{e_m/x_m\}) = e\{s_1/x_1\} \dots \{s_m/x_m\}$*

Proof: By case analysis. The only interesting case is $e = x$:

By the assumption $\text{fv}(x\{e_1/x_1\} \dots \{e_m/x_m\}) = \emptyset$, we know that $x = x_i$ for some $i \in 1..m$ and either $e_i = s$ or $e_i = \mathbf{sizeof}(t')$. Then, by inversion of the evaluation judgment, $t; \delta; \tau'; 0 \vdash e_i \Downarrow s_i$ we know that either $e_i = s_i$ or $e_i = \mathbf{sizeof}(t')$, where $\rho_{sz}(t') = s_i$. Hence, $\rho_{sz}(e_i) = s_i$, as required.

■

Lemma 126 *If $(\rho_{sz}; \delta; \tau'; \eta) \models_{\rho_{sz}(t)}^{int} \Gamma$ and $t; \delta; \tau'; 0 \vdash e_k \Downarrow s_k$ for all $k \in 1..m$ and T contains no indexed expressions of the form **sizeof**(t') and $\{x_1, \dots, x_m\} \cap (\bigcup_k \text{fv}(e_k)) = \emptyset$ and $\text{fv}(T\{e_1/x_1\} \dots \{e_m/x_m\}) = \emptyset$, then $\rho_{sz}(T\{e_1/x_1\} \dots \{e_m/x_m\}) = T\{s_1/x_1\} \dots \{s_m/x_m\}$*

Proof: By induction on the structure of T , with appeal to Lemma 125. ■

The below lemma states that evaluating a well-typed expression yields a well-typed value:

Lemma 127 *If $\Gamma, \Gamma' \vdash^{pc} E : T$ and $\neg \text{rnd}(T)$ and $(\rho_{sz}; \delta; \tau'; \eta) \models_{\rho_{sz}(t)}^{loc} (\Gamma; \Gamma')$ and $t; \delta; \tau'; i \vdash E \Downarrow G$ for some $i \in 0..\rho_{sz}(t) - 1$, then $\emptyset \vdash^{pc} G : \rho_{sz}(\tau'(T))$, where $\text{value}(G)$ if $\text{det}(T)$.*

Proof:

By induction on the derivation of $\Gamma, \Gamma' \vdash^{pc} E : T$. Interesting cases:

- Case (DEREF INST);

(DEREF INST)

$\Gamma, \Gamma' \vdash^{pc} E' : \text{link}(t) ! \text{spc}$

$\Gamma, \Gamma' = \Gamma'_1, t : Q, \Gamma''_1 \quad Q = Q' @ [(c \triangleright x : T' \text{ inst viz})] @ Q''$

$\Gamma, \Gamma' \vdash^{pc} E' : t.c : T' \vee \text{spc}$

Here, $t; \delta; \tau'; i \vdash E \Downarrow G$ must have been derived with (QUERY Deref INST) or (QUERY Deref INST FAIL).

- Subcase (QUERY Deref INST):

(QUERY Deref INST)

$t; \delta; \tau'; i \vdash E' \Downarrow k$

$k \in 0..\rho_{sz}(t) - 1$

$\delta(t)(c) = \text{inst}[G_0, \dots, G_{\rho_{sz}(t)-1}]$

$t; \delta; \tau'; i \vdash E' : t.c \Downarrow G_k$

We have $t; \delta; \tau'; i \vdash E' \Downarrow k$ and $\delta(t)(c) = \text{inst}([G_0, \dots, G_{\rho_{sz}(t)-1}])$, where $G = G_k$.

By Lemma 122, $\delta = \delta_1 @ [(t \mapsto \tau_t)] @ \delta_2$ and $\tau_t \models_{\rho_{sz}(t)}^{out} Q$. Since $Q = Q' @ [(c \triangleright x : T' \text{ inst viz})] @ Q''$, by Lemma 123 we have

$\tau_t = \delta(t) = \tau_1 @ [(c \mapsto \text{inst}([G_0, \dots, G_{\rho_{sz}(t)-1}]))] @ \tau_2$ and $\emptyset \vdash^{\text{inst}} G_i : \rho_{sz}(T')$

for all $i \in 0..\rho_{sz}(t) - 1$ and $\text{value}(G_i)$ for all $i \in 0..\rho_{sz}(t) - 1$ if $\text{det}(T')$,

which implies $\emptyset \vdash^{\text{inst}} G_k : \rho_{sz}(T')$ and $\text{value}(G_k)$ if $\text{det}(T')$. The predicate

table(Q), which holds by the well-formedness of Γ , implies $\text{fv}(T') = \emptyset$,

so $\tau'(T') = T'$. Since, obviously, $\Gamma \vdash T' <: T' \vee \text{spc}$, by (SUBSUM) we have $\emptyset \vdash^{\text{inst}} G_k : \rho_{sz}(T' \vee \text{spc})$, and so $\emptyset \vdash^{\text{inst}} G_k : \rho_{sz}(\tau'(T' \vee \text{spc}))$, where $\text{value}(G_k)$ if $\text{det}(T' \vee \text{spc})$, as required.

– Subcase (QUERY Deref INST FAIL):

(QUERY Deref INST FAIL)

$$\frac{t; \delta; \tau'; i \vdash E' \Downarrow \text{fail}}{t; \delta; \tau'; i \vdash E' : t.c \Downarrow \text{fail}}$$

By assumption, fail checks against any type, so we have $\emptyset \vdash^{pc} \text{fail} : \rho_{sz}(\tau'(T))$. By induction hypothesis, we have $\neg \text{det}(T)$, as required.

• Case (ITER):

(ITER) (where $x \notin \text{fv}(T')$)

$\Gamma, \Gamma' \vdash^{\text{static}} e : \text{int} ! \text{det}$

$\Gamma, \Gamma', x :^{pc} (\text{mod}(e) ! \text{det}) \vdash^{pc} F : T'$

$\Gamma, \Gamma' \vdash^{pc} [\text{for } x < e \rightarrow F] : T'[e]$

The judgment $t; \delta; \tau'; i \vdash E \Downarrow G$ must have been derived with (QUERY ITER):

(QUERY ITER)

$$\frac{t; \delta; \tau'; i \vdash e \Downarrow m \quad t; \delta; \tau', (x \mapsto \text{static}(j)); i \vdash F \Downarrow G_j \quad \forall j \in 0..m-1}{t; \delta; \tau'; i \vdash [\text{for } x < e \rightarrow F] \Downarrow [G_0, \dots, G_{m-1}]}$$

By Lemma 124 we have $\rho_{sz}(\tau'(e)) = m$. Since $\emptyset \vdash^{\text{static}} j : \text{mod}(m) ! \text{det}$ for all $j \in 0..m-1$, by (CONF VAR LOCAL), $(\rho_{sz}; \delta; \tau', (x \mapsto \text{static}(j)); \eta) \models_n^{loc} (\Gamma; \Gamma', x :^{pc} (\text{mod}(e) ! \text{det}))$. By induction hypothesis, $\emptyset \vdash^{pc} m : \text{int} ! \text{det}$ and $\emptyset \vdash^{pc} G_j : \rho_{sz}(\tau'(T'))$ for all $j \in 0..m-1$, where $\text{value}(G_j)$ for all $j \in 0..m-1$ if $\text{det}(T)$. Hence, by (ARRAY), we have $\Gamma \vdash^{pc} [G_0, \dots, G_{m-1}] : \rho_{sz}(\tau'(T'))[m]$, where $\text{value}([G_0, \dots, G_{m-1}])$ if $\text{det}(T'[m])$, as required.

• Case (INDEX SIZEOF):

(INDEX SIZEOF)

$\Gamma, \Gamma' \vdash \diamond \quad \Gamma, \Gamma' = \Gamma^*, t : Q, \Gamma^{**}$

$\Gamma, \Gamma' \vdash^{pc} \text{sizeof}(t) : \text{int} ! \text{det}$

In this case $t; \delta; \tau'; i \vdash E \Downarrow G$ must have been derived with (QUERY SIZEOF):

(QUERY SIZEOF)

$$\frac{\rho_{sz}(t) = n}{t; \delta; \tau'; i \vdash \text{sizeof}(t) \Downarrow n}$$

We have $V = \rho_{sz}(t)$. By Lemma 122, $\rho_{sz}(t) \in \mathbb{N}$, so we have $\emptyset \vdash^{pc} \rho_{sz}(t) : \mathbf{int} ! \mathbf{det}$.

- Case (INDEX VAR):

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

$$\frac{\Gamma, \Gamma' \vdash \diamond \quad \Gamma, \Gamma' = \Gamma_1, x : ^\ell T, \Gamma_2}{\Gamma, \Gamma' \vdash^{pc} x : T}$$

In this case, $t; \delta; \tau'; i \vdash E \Downarrow G$ must have been derived with either (QUERY VAR STATIC) or (QUERY VAR INST):

- Subcase (QUERY VAR STATIC):

(QUERY VAR STATIC)

$$\frac{\tau(x) = \mathbf{static}(G)}{t; \delta; \tau; i \vdash x \Downarrow G}$$

- * If $x \in \text{dom}(\Gamma)$ and $\ell = \mathbf{static}$, then by Lemma 120 (part 1), $\tau'(x) = \mathbf{static}(G')$ and $\emptyset \vdash^{\mathbf{static}} G' : \rho_{sz}(\tau'(T))$ for some G' such that $\text{value}(G')$ if $\mathbf{det}(T)$. Since identifiers in τ' are unique, $G = G'$, so $\emptyset \vdash^{\mathbf{static}} G : \rho_{sz}(\tau'(T))$ and $\text{value}(G)$ if $\mathbf{det}(T)$ as required.
- * If $x \in \text{dom}(\Gamma)$ and $\ell = \mathbf{inst}$, then by Lemma 120 (part 2), $\tau'(x) = \mathbf{inst}([G_0, \dots, G_{\rho_{sz}(t)-1}])$, which contradicts the assumption that identifiers in τ' are unique, so this case is not possible.
- * If $x \in \text{dom}(\Gamma')$, then by Lemma 120 (part 3), $\tau'(x) = \mathbf{static}(s)$ and $\emptyset \vdash^{\mathbf{static}} s : \rho_{sz}(\tau'(T))$ for some scalar s . Like in the first case, we have $G = s$, so $\emptyset \vdash^{\mathbf{static}} s : \rho_{sz}(\tau'(T))$, as required.

- Subcase (QUERY VAR INST):

(QUERY VAR INST)

$$\frac{\tau(x) = \mathbf{inst}([G_0, \dots, G_{\rho_{sz}(t)-1}])}{t; \delta; \tau; i \vdash x \Downarrow V_i}$$

- * If $x \in \text{dom}(\Gamma)$ and $\ell = \mathbf{static}$, then by Lemma 120 (part 1), $\tau'(x) = \mathbf{static}(G')$, which contradicts the assumption that identifiers in τ' are unique. Hence, this case is not possible.
- * If $x \in \text{dom}(\Gamma)$ and $\ell = \mathbf{inst}$, then by Lemma 120 (part 2), $\tau'(x) = \mathbf{inst}([G'_0, \dots, G'_{\rho_{sz}(t)-1}])$ and $\emptyset \vdash^{\mathbf{inst}} G'_j : \rho_{sz}(\tau'(T))$ for all $j \in 0.. \rho_{sz}(t) - 1$. and $\text{value}(G'_j)$ for all $j \in 0.. \rho_{sz}(t) - 1$ if $\mathbf{det}(T)$. Again, since identifiers in τ' are unique, $G'_i = G_i$. Hence, $\emptyset \vdash^{\mathbf{inst}} G_i : \rho_{sz}(\tau'(T))$.

- * If $x \in \text{dom}(\Gamma')$, then by Lemma 120 (part 3), $\tau'(x) = \mathbf{static}(G')$, which, again, contradicts the uniqueness of identifiers.

■

This yields the following result for top-level expressions in table columns:

Corollary 7 *If $\Gamma \vdash^{pc} E : T$ and $\text{fv}(T) = \emptyset$ and $\neg \mathbf{rnd}(T)$ and $(\rho_{sz}, \delta, \tau', \eta) \models_n^{int} (\Gamma)$ and $t; \delta; \tau'; i \vdash E \Downarrow G$ for some $i \in 0..n-1$ then $\emptyset \vdash^{pc} G : \rho_{sz}(\tau(T))$, where $\text{value}(G)$ if $\mathbf{det}(T)$.*

Lemma 128 *For any distribution signature $D_{spc} : [x_1 : T_1, \dots, x_m : T_m](c_1 : U_1, \dots, c_n : U_n) \rightarrow T'$, the types T_1, \dots, T_m have no free variables and **sizeof** expressions, and types U_1, \dots, U_n have no **sizeof** expressions and for all $i \in 1..n$, $\text{fv}(U_i) \subseteq \{x_1, \dots, x_n\}$.*

Proof: By inspection of the signatures of distributions. ■

All table-level databases obtained by evaluating well-typed tables conform to the table types:

Lemma 129 *If $\Gamma \vdash^{inst} \mathbb{T} : Q$ and $\text{Core}(\mathbb{T})$ and $\mathbf{table}(Q)$ and $\text{fv}(Q) = \emptyset$ and $(\delta_{in}(t), \rho_{sz}) \models_{\rho_{sz}(t)} Q$ and $t; \delta_{in}; \delta; \tau'; \eta \vdash \mathbb{T} \Downarrow \tau$ and $(\rho_{sz}, \delta, \tau', \eta) \models_{\rho_{sz}(t)}^{int} \Gamma$, then $(\tau, \rho_{sz}) \models_{\rho_{sz}(t)}^{out} Q$.*

Proof: By induction on the derivation of $t; \delta_{in}; \delta; \tau'; \eta \vdash \mathbb{T} \Downarrow \tau$:

- Case (VAL QUERY STATIC):

(VAL QUERY STATIC)

$t; \delta; \tau'; 0 \vdash e_k \Downarrow s_k \quad \forall k \in 1..m$

$(G_1, \dots, G_n) = \arg \min_{y_1, \dots, y_n} \|D[s_1, \dots, s_m](y_1, \dots, y_n) - \eta(c)\|$

$t; \delta_{in}; \delta; \tau', (x \rightarrow \mathbf{static}(G_j)); \eta \vdash \mathbb{T} \Downarrow \hat{\tau}$

$t; \delta_{in}; \delta; \tau'; \eta \vdash (c \triangleright x : T \mathbf{static} \text{ viz } (\mathbf{infer}.D[e_1, \dots, e_m].c_j(E')) :: \mathbb{T}' \Downarrow \hat{\tau}@[c \mapsto \mathbf{static}(G_j)])$

- Subcase $\text{viz} = \mathbf{output}$:

In this case, $\Gamma \vdash^{inst} \mathbb{T} : Q$ must have been derived with (TABLE CORE OUTPUT):

(TABLE CORE OUTPUT)

$\Gamma \vdash^{\mathbf{static}} E : T \quad \Gamma, x : \mathbf{static} T \vdash^{inst} \mathbb{T}' : Q' \quad c \notin \text{names}(Q)$

$\Gamma \vdash^{inst} (c \triangleright x : T \mathbf{static} \text{ output } E) :: \mathbb{T}' : (c \triangleright x : T \mathbf{static} \text{ output}) :: Q'$

where $E = \mathbf{infer}.D[e_1, \dots, e_m].c_j(E')$.

Hence, we have $\Gamma \vdash^{\mathbf{static}} \mathbf{infer}.D[e_1, \dots, e_m].c_j(E') : T$. This judgment must have been derived with (INFER):

$$\begin{array}{l} \text{(INFER) (where } \sigma(U) \triangleq U\{e_1/x_1\} \dots \{e_m/x_m\}) \\ D_{\mathbf{qry}} : [x_1 : T_1, \dots, x_m : T_m](c_1 : U_1, \dots, c_n : U_n) \rightarrow T' \\ \Gamma \vdash^{\mathbf{static}} e_i : T_i \quad \forall i \in 1..m \quad \Gamma \vdash^{\mathbf{static}} E' : \sigma(T') \quad j \in 1..n \\ \{x_1, \dots, x_m\} \cap (\bigcup_i \text{fv}(e_i)) = \emptyset \quad x_i \neq x_j \text{ for } i \neq j \\ \hline \Gamma \vdash^{\mathbf{static}} \mathbf{infer}.D[e_1, \dots, e_m].c_j(E') : \sigma(U_j) \end{array}$$

followed by 0 or more applications of (SUBSUM). Hence, we have $\Gamma \vdash \sigma(U_j) <: T$, which in fact implies $\sigma(U_j) = T$, as U_j is in **qry**-space by inspection of the distribution signatures.

By inspection of the distribution signatures, $\text{fv}(T_k) = \emptyset$, so by Corollary 7 and Lemma 128, we have $\emptyset \vdash^{pc} s_k : \rho_{sz}(T_k)$.

As the $\arg \min$ operator takes a minimum over well-typed values (or returns a tuple of exceptions `fail`, which check against any type), we have $\emptyset \vdash^{\mathbf{static}} G_j : U_j \{s_1/x_1\} \dots \{s_m/x_m\}$.

As $\sigma(U_j) = T$ is a top-level column type, it can, by assumption, contain no free variables. By Lemma 128, U_j contains no table size references.

Thus, by Lemma 126, we have $\rho_{sz}(T) = \rho_{sz}(\sigma(U_j)) = U_j \{s_1/x_1\} \dots \{s_1/x_1\}$, so $\emptyset \vdash^{\mathbf{static}} G_j : \rho_{sz}(T)$.

By (CONF VAR STATIC), we have $(\rho_{sz}; \delta; \tau', (x \mapsto \mathbf{static}(G_j)); \eta) \models_{\rho_{sz}(t)}^{int} \Gamma, x : \mathbf{static} T$ (recall that $T = \sigma(U_j)$ is not **det**, so G_j can be `fail`). Moreover, $(\delta_{in}(t), \rho_{sz}) \models_{\rho_{sz}(t)} (c \triangleright x : T \mathbf{static output}) :: Q'$ must have been derived with (CONF STATIC OUTPUT), which implies $(\delta_{in}(t), \rho_{sz}) \models_{\rho_{sz}(t)} Q'$. Hence, by induction hypothesis, $(\hat{\tau}, \rho_{sz}) \models_{\rho_{sz}(t)}^{out} Q'$. Therefore, by (CONF STATIC OUTPUT OUT), $(\hat{\tau}, (c \mapsto \mathbf{static}(G_j)); \rho_{sz}) \models_{\rho_{sz}(t)}^{out} (c \triangleright x : T \mathbf{static output}) :: Q'$.

– Subcase $vi_z = \mathbf{local}$: similar.

- Case (VAL QUERY INST): similar to the previous case (with the reasoning repeated for every index).
- Case (VAL QUERYORDET STATIC):

$$\begin{array}{c}
(\text{VAL QUERYORDET STATIC}) \text{ (where } \text{space}(T) \neq \mathbf{rnd}, E \neq \mathbf{infer}.D[e_1, \dots, e_m].c_j(E')) \\
t; \delta; \tau'; 0 \vdash E \Downarrow G \\
t; \delta_{in}; \delta; \tau', (x \rightarrow \mathbf{static}(G)); \eta \vdash \mathbb{T}' \Downarrow \hat{\tau} \\
\hline
t; \delta_{in}; \delta; \tau'; \eta \vdash (c \triangleright x : T \mathbf{static} \text{ viz } E) :: \mathbb{T}' \Downarrow \hat{\tau}@[c \mapsto \mathbf{static}(G)]
\end{array}$$

– Subcase $\text{viz} = \mathbf{output}$:

Here, $\Gamma \vdash^{\mathbf{inst}} \mathbb{T} : Q$ must have been derived with (TABLE CORE OUTPUT):

(TABLE CORE OUTPUT)

$$\begin{array}{c}
\Gamma \vdash^{\mathbf{static}} E : T \quad \Gamma, x :^{\mathbf{static}} T \vdash^{\mathbf{inst}} \mathbb{T}' : Q' \quad c \notin \text{names}(Q) \\
\hline
\Gamma \vdash^{\mathbf{inst}} (c \triangleright x : T \mathbf{static} \text{ output } E) :: \mathbb{T}' : (c \triangleright x : T \mathbf{static} \text{ output}) :: Q'
\end{array}$$

By Corollary 7, we have $\emptyset \vdash^{\mathbf{static}} G : \rho_{sz}(\tau(T))$ and $\text{value}(G)$ if $\mathbf{det}(T)$. As $\text{fv}(T) = \emptyset$, this implies $\emptyset \vdash^{\mathbf{static}} G : \rho_{sz}(T)$. By (CONF VAR STATIC), $(\rho_{sz}; \delta; \tau', (x \mapsto \mathbf{static}(G)); \eta) \models_{\rho_{sz}(t)}^{\text{int}} \Gamma, x :^{\mathbf{static}} T$. By the derivation of $(\delta_{in}(t), \rho_{sz}) \models_{\rho_{sz}(t)} (c \triangleright x : T \mathbf{static} \text{ output}) :: Q'$ we have $(\delta_{in}(t), \rho_{sz}) \models_{\rho_{sz}(t)} Q'$. By induction hypothesis, $(\hat{\tau}; \rho_{sz}) \models_{\rho_{sz}(t)}^{\text{out}} Q'$. Thus, by (CONF STATIC OUTPUT OUT), $(\hat{\tau}, (c \mapsto \mathbf{static}(G)); \rho_{sz}) \models_{\rho_{sz}(t)}^{\text{out}} (c \triangleright x : T \mathbf{static} \text{ output}) :: Q'$, as required.

– Subcase $\text{viz} = \mathbf{local}$: similar.

- Case (VAL QUERYORDET INST): similar to (VAL QUERYORDET STATIC).

- Case (VAL EMPTY):

(VAL EMPTY)

$$\begin{array}{c}
\hline
t; \delta_{in}; \delta; \tau'; \eta \vdash [] \Downarrow []
\end{array}$$

Here, $\Gamma \vdash^{\mathbf{inst}} \mathbb{T} : Q$ must have been derived with (TABLE []), so we have $([], \rho_{sz}) \models_{\rho_{sz}(t)}^{\text{out}}$ [] by (CONF [] OUT).

- Case (VAL INPUT):

(VAL INPUT) (where $\text{space}(T) \neq \mathbf{rnd}$)

$$\begin{array}{c}
t; \delta_{in}; \delta; \tau', (x \rightarrow \delta_{in}(t)(c)); \eta \vdash \mathbb{T}' \Downarrow \tau \\
\hline
t; \delta_{in}; \delta; \tau'; \eta \vdash (c \triangleright x : T \ell \mathbf{input} \varepsilon) :: \mathbb{T}' \Downarrow \tau@[c \mapsto \delta_{in}(t)(c)]
\end{array}$$

Here, $\Gamma \vdash^{\mathbf{inst}} \mathbb{T} : Q$ must have been derived with (TABLE INPUT), so $Q = (c \triangleright x : T \ell \mathbf{input}) :: Q'$ for some Q' .

If $\ell = \mathbf{static}$, then $(\delta_{in}(t), \rho_{sz}) \models_{\rho_{sz}(t)} (c \triangleright x : T \mathbf{static} \text{ input}) :: Q'$ must have been derived with (CONF STATIC INPUT), so $(\delta_{in}(t), \rho_{sz}) \models_{\rho_{sz}(t)} Q'$ and $\delta_{in}(t)(c) =$

static(V) for some V such that $\emptyset \vdash^{\text{static}} V : \rho_{sz}(T)$ (note that V cannot be **fail**, as there are no exceptions in the input database). By the derivation of $(\delta_{in}(t), \rho_{sz}) \models_{\rho_{sz}(t)}$ $(c \triangleright x : T \text{ static input}) :: Q'$ we have $(\delta_{in}(t), \rho_{sz}) \models_{\rho_{sz}(t)} Q'$. Thus, by induction hypothesis, $(\tau'; \rho_{sz}) \models_{\rho_{sz}(t)}^{out} Q'$, so by (CONF STATIC INPUT OUT) we have $(\tau', (c \mapsto \text{static}(V)); \rho_{sz}) \models_{\rho_{sz}(t)}^{out} (c \triangleright x : T \ell \text{ input}) :: Q'$,

If $\ell = \text{inst}$, then $(\delta_{in}(t), \rho_{sz}) \models_{\rho_{sz}(t)} (c \triangleright x : T \text{ inst input}) :: Q'$ must have been derived with (CONF INST INPUT), so $(\delta_{in}(t), \rho_{sz}) \models_{\rho_{sz}(t)} Q'$ and $\delta_{in}(t)(c) = \text{inst}([V_0, \dots, V_{\rho_{sz}(t)-1}])$ for some V such that $\emptyset \vdash^{\text{static}} V_i : \rho_{sz}(T)$ for all $i \in 0.. \rho_{sz}(t) -$

1. By the derivation of $(\delta_{in}(t), \rho_{sz}) \models_{\rho_{sz}(t)} (c \triangleright x : T \text{ inst input}) :: Q'$ we have $(\delta_{in}(t), \rho_{sz}) \models_{\rho_{sz}(t)} Q'$. Thus, by induction hypothesis, $(\tau'; \rho_{sz}) \models_{\rho_{sz}(t)}^{out} Q'$, so by (CONF INST INPUT OUT) we have $(\tau', (c \mapsto \text{inst}([V_0, \dots, V_{\rho_{sz}(t)-1}])); \rho_{sz}) \models_{\rho_{sz}(t)}^{out} (c \triangleright x : T \ell \text{ input}) :: Q'$,

- Case (VAL RANDOM):

(VAL RANDOM) (where $\text{space}(T) = \text{rnd}$)

$$\frac{t; \delta_{in}; \delta; \tau'; \eta \vdash \mathbb{T}' \Downarrow \tau}{t; \delta_{in}; \delta; \tau'; \eta \vdash (c \triangleright x : T \ell \text{ viz } M) :: \mathbb{T}' \Downarrow \tau}$$

- Case $\text{viz} = \text{output}$

In this case, $\Gamma \vdash^{\text{inst}} (c \triangleright x : T \ell \text{ output } M) :: \mathbb{T}' : Q$ must have been derived with (TABLE CORE OUTPUT), so $Q = (c \triangleright x : T \ell \text{ output}) :: Q'$ for some Q' such that $\Gamma, x : {}^\ell T \vdash^{\text{inst}} \mathbb{T}' : Q'$. By (CONF VAR RND), $(\rho_{sz}; \delta; \tau'; \eta) \models_{\rho_{sz}(t)}^{\text{int}} \Gamma, x : {}^\ell T$. By the derivation of $(\delta_{in}(t); \rho_{sz}) \models_{\rho_{sz}(t)} (c \triangleright x : T \ell \text{ output}) :: Q'$ we have $(\delta_{in}(t); \rho_{sz}) \models_{\rho_{sz}(t)} Q'$. Hence, $(\tau'; \rho_{sz}) \models_{\rho_{sz}(t)}^{out} (c \triangleright x : T \ell \text{ output}) :: Q'$, follows by the induction hypothesis.

- Cases $\text{viz} = \text{input}$ and $\text{viz} = \text{local}$ are similar.

■

Finally, we obtain the conformance result for schemas:

Lemma 130 *If $\text{Core}(\mathbb{S})$ and $\Gamma \vdash \mathbb{S} : \text{Sty}$ and Γ only contains entries of the form $(t : Q')$ and $(\rho_{sz}; \delta; []; \eta) \models^{\text{int}} \Gamma$ and $(\delta_{in}; \rho_{sz}) \models \text{Sty}$ and $\delta_{in}; \delta \vdash_{\sigma} \mathbb{S} \Downarrow \delta_{out}$, then $(\delta_{out}; \rho_{sz}) \models^{out} \text{Sty}$.*

Proof: By induction on the derivation of $\Gamma \vdash \mathbb{S} : \text{Sty}$:

- Case (SCHEMA TABLE):

(SCHEMA TABLE)

$$\frac{\Gamma \vdash^{\text{inst}} \mathbb{T} : Q \quad \text{table}(Q) \quad \Gamma, t : Q \vdash \mathbb{S}' : \text{Sty}'}{\Gamma \vdash (t = \mathbb{T}) :: \mathbb{S}' : (t : Q) :: \text{Sty}'}$$

Since $(\delta_{in}; \rho_{sz}) \models \text{Sty}$ and $\delta_{in}, \delta \vdash_{\sigma} \mathbb{S} \Downarrow \delta_{out}$ must have been derived with (CONF SCHEMA TABLE) and (QUERY SCHEMA TABLE) respectively, from inversion of these rules we get $\rho_{sz}(t) \in \mathbb{N}$ and $(\delta_{in}(t); \rho_{sz}) \models_{\rho_{sz}(t)} Q$ and $(\delta_{in}; \rho_{sz}) \models \text{Sty}'$ and $t; \delta_{in}; \delta; \emptyset; \sigma(t) \vdash \mathbb{T} \Downarrow \tau$ and $\delta_{in}; \delta, (t \rightarrow \tau) \vdash_{\sigma} \mathbb{S}' \Downarrow \delta'_{out}$. where $\delta_{out} = (t \mapsto \tau) :: \delta'_{out}$.

As Γ only contains table entries, we have $\text{fv}(Q) = \emptyset$. By Lemma 129, we have $(\tau; \rho_{sz}) \models_{\rho_{sz}(t)}^{out} Q$, so by (CONF SCHEMA TABLE), $(\rho_{sz}; \delta, (t \mapsto \tau); \emptyset; \eta) \models^{int} \Gamma, t : Q$. By induction hypothesis, $(\delta'_{out}; \rho_{sz}) \models^{out} \text{Sty}'$, so by (CONF SCHEMA TABLE OUT) we have $((t \mapsto \tau) :: \delta'_{out}; \rho_{sz}) \models^{out} (t = Q) :: \text{Sty}'$, as required.

- Case (SCHEMA \square):

(SCHEMA \square)

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \square : \square}$$

The result follows immediately by (CONF \square OUT).

■

Restatement of Lemma 14 *If $\text{Core}(\mathbb{S})$ and $\emptyset \vdash \mathbb{S} : \text{Sty}$ and $(\delta_{in}; \rho_{sz}) \models \text{Sty}$ and $\delta_{in}; \square \vdash_{\sigma} \mathbb{S} \Downarrow \delta_{out}$, then $(\delta_{out}; \rho_{sz}) \models^{out} \text{Sty}$.*

Proof: Corollary of Lemma 130. ■

C.3 Progress Result for Query Semantics

Finally, we show that every well-typed schema with a conforming input database and marginal map actually evaluates to an output database.

Restatement of Lemma 13 *If $\emptyset \vdash \mathbb{S} : \text{Sty}$ and $\text{Core}(\mathbb{S})$ and $(\delta_{in}; \rho_{sz}) \models \text{Sty}$ and $(\sigma, \rho_{sz}) \models^{marg} \mathbb{S}$ then $\delta_{in}, \square \vdash \mathbb{S} \Downarrow_{\sigma} \delta_{out}$ for some δ_{out} .*

As usual, we need some auxiliary results:

Lemma 131 *If $\text{Core}(\mathbb{S})$ and $(\sigma; \rho_{sz}) \models^{marg} \mathbb{S}$ and $\mathbb{S} = \mathbb{S}', (t \mapsto \mathbb{T}), \mathbb{S}'$, then $(\sigma(t); \rho_{sz}) \models^{marg} \mathbb{T}$.*

Proof: Follows immediately from the marginal map conformance rules. ■

Lemma 132 (1) *If $(\rho_{sz}, \delta, \tau', \eta) \models_n^{int} \Gamma', x : \mathbf{static} T, \Gamma''$ and $\text{rnd}(T)$, then $\eta(x) = \mathbf{static}(v)$*

(2) *If $(\rho_{sz}, \delta, \tau', \eta) \models_n^{int} \Gamma', x : \mathbf{inst} T, \Gamma''$ and $\text{rnd}(T)$, then $\eta(x) = \mathbf{inst}([v_0, \dots, v_{n-1}])$*

Proof:

(1) By induction on the derivation of $(\rho_{sz}, \delta, \tau', \eta) \models_n^{int} \Gamma', x : \mathbf{static} T, \Gamma''$

(2) By induction on the derivation of $(\rho_{sz}, \delta, \tau', \eta) \models_n^{int} \Gamma', x : \mathbf{inst} T, \Gamma''$

■

Lemma 133 *If $\Gamma, \Gamma' \vdash^{\mathbf{static}} e : \mathbf{int} ! \mathbf{det}$ and $(\rho_{sz}; \delta; \tau'; \eta) \models_n^{loc} (\Gamma; \Gamma')$, then for every $i \in 0..n-1$, $\delta; \tau'; \eta; i \vdash e \Downarrow G$ for some unique G such that $\text{value}(G)$.*

Proof: By case analysis on the structure of e :

- If $e = c$, the result is obvious.
- If $e = \mathbf{sizeof}(t)$, then $\Gamma, \Gamma' \vdash^{\mathbf{static}} e : \mathbf{int} ! \mathbf{det}$ must have been derived with (INDEX SIZEOF), so $\Gamma, \Gamma' = \Gamma_1, t : Q, \Gamma_2$. By Lemma 122, we have $\rho_{sz}(t) \in \mathbb{N}$, so by (QUERY SIZEOF), we have $\delta, \tau', \eta, i \vdash e \Downarrow \rho_{sz}(t)$ for every $i \in 0..n-1$.
- If $e = x$, then $\Gamma, \Gamma' \vdash^{\mathbf{static}} e : \mathbf{int} ! \mathbf{det}$ must have been derived with (INDEX VAR) followed by zero or more applications of (SUBSUM). Hence, $\Gamma, \Gamma' = \Gamma_1, x : \mathbf{static} U, \Gamma_2$, where $\Gamma \vdash U <: \mathbf{int} ! \mathbf{det}$, which implies $U = \mathbf{int} ! \mathbf{det}$.

If $x \in \text{dom}(\Gamma)$, then by Lemma 120 (part 1), we have $\tau'(x) = \mathbf{static}(G)$ and $\text{value}(G)$. If $x \in \text{dom}(\Gamma')$, we also have $\tau'(x) = \mathbf{static}(G)$ and $\text{value}(G)$ by part 3 of Lemma 120. In either case, we have $\delta, \tau', \eta, i \vdash e \Downarrow G$ by (QUERY VAR STATIC) and since identifiers in τ' are unique, there is no other derivation, so the G is unique.

We now prove the progress result for expressions: every well-typed expression in an evaluation environment conforming to the type environment evaluates to a value (which may, in general, be fail).

Lemma 134 *If $\Gamma, \Gamma' \vdash^{pc} E : T$ and $\neg \text{rnd}(T)$ and $(\rho_{sz}; \delta; \tau'; \eta) \models_n^{loc} (\Gamma; \Gamma')$ then for all $i \in 0..n-1$, $\delta; \tau'; \eta; i \vdash E \Downarrow G_i$ for some G_i .*

Proof: By induction on the derivation of $\Gamma \vdash^{pc} E : T$. Interesting cases:

- Case (DEREF INST):

(DEREF INST)

$\Gamma, \Gamma' \vdash^{pc} E' : \text{link}(t) ! \text{spc}$

$\frac{\Gamma, \Gamma' = \Gamma_1, t : Q, \Gamma_2 \quad Q = Q' @ [(c \triangleright x : T' \text{ inst viz})] @ Q''}{\Gamma, \Gamma' \vdash^{pc} E' : t.c : T' \vee \text{spc}}$

$\Gamma, \Gamma' \vdash^{pc} E' : t.c : T' \vee \text{spc}$

By induction hypothesis, we have $\delta, \tau', \eta, i \vdash E' \Downarrow G'_i$ for all i . By Lemma 127, we have $\emptyset \vdash^{pc} G'_i : \text{mod}(\rho_{sz}(t)) ! \text{spc}$, so by inversion of typing we have either $G'_i = n_i$ (where $0 \leq n_i < \rho_{sz}(t) - 1$) or $G'_i = \text{fail}$.

By Lemmas 122 and 123, we have $\delta(t)(c) = \text{inst}([G_0, \dots, G_{\rho_{sz}(t)-1}])$, so for all $i \in 0..n-1$, we have $\delta, \tau', \eta, i \vdash E' : t.c \Downarrow G_{n_i}$ by (QUERY DEREF INST) or $\delta, \tau', \eta, i \vdash E' : t.c \Downarrow \text{fail}$ by (QUERY DEREF INST FAIL).

- Case (ITER):

(ITER) (where $x \notin \text{fv}(T')$)

$\Gamma, \Gamma' \vdash^{\text{static}} e : \text{int} ! \text{det}$

$\frac{\Gamma, \Gamma', x :^{pc} (\text{mod}(e) ! \text{det}) \vdash^{pc} F : T'}{\Gamma, \Gamma' \vdash^{pc} [\text{for } x < e \rightarrow F] : T'[e]}$

$\Gamma, \Gamma' \vdash^{pc} [\text{for } x < e \rightarrow F] : T'[e]$

By Lemma 133, for any i , we have $\delta, \tau', \eta, i \vdash e \Downarrow G$ for some unique G such that $\text{value}(G)$. By Lemma 127, $\emptyset \vdash^{pc} G : \text{int} ! \text{det}$, so $G = m$ for some integer m .

By Lemma 124 we have $\rho_{sz}(\tau'(e)) = m$, so by (CONF VAR LOCAL), $(\rho_{sz}; \delta; \tau', (x \mapsto \text{static}(j)); \eta) \models_n^{\text{int}} (\Gamma; \Gamma', x :^{pc} (\text{mod}(e) ! \text{det}))$ for all $j \in 0..m-1$. Hence, by induction hypothesis, for every i we have $\delta; \tau', (x \mapsto \text{static}(j)); \eta; i \vdash F \Downarrow G_j$ for some G_j for every $j \in 0..m-1$. Thus, by (QUERY ITER), we have $\delta, \tau', \eta, i \vdash e \Downarrow [G_0, \dots, G_{m-1}]$ for all $i \in 0..n-1$.

- Case (INDEX SIZEOF):

(INDEX SIZEOF)

$$\frac{\Gamma, \Gamma' \vdash \diamond \quad \Gamma, \Gamma' = \Gamma_1, t : Q, \Gamma_2}{\Gamma, \Gamma' \vdash^{pc} \mathbf{sizeof}(t) : \mathbf{int} ! \mathbf{det}}$$

By Lemma 122, we have $\rho_{sz}(t) \in \mathbb{N}$, so by (QUERY SIZEOF), we have $\delta, \tau', \eta, i \vdash E \Downarrow \rho_{sz}(t)$ for every $i \in 0..n-1$.

- Case (INDEX VAR):

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

$$\frac{\Gamma, \Gamma' \vdash \diamond \quad \Gamma, \Gamma' = \Gamma_1, x : ^\ell T, \Gamma_2}{\Gamma, \Gamma' \vdash^{pc} x : T}$$

- If $x \in \text{dom}(\Gamma)$ and $\ell = \mathbf{static}$, then by Lemma 120, we have $\tau'(x) = \mathbf{static}(G)$, so we get the required result by (QUERY VAR STATIC).
- If $x \in \text{dom}(\Gamma)$ and $\ell = \mathbf{static}$, then by Lemma 120, we have $\tau'(x) = \mathbf{inst}([G_0, \dots, G_{n-1}])$, so by (QUERY VAR STATIC) we get $\delta, \tau', \eta, i \vdash E \Downarrow G_i$ for all $i \in 0..n-1$.
- If $x \in \text{dom}(\Gamma')$, then by Lemma 120, we have $\tau'(x) = \mathbf{static}(G)$, so the desired result follows by (QUERY VAR STATIC).

■

Corollary 8 *If $\Gamma \vdash^{pc} E : T$ and $\neg \mathbf{rnd}(T)$ and $(\rho_{sz}; \delta; \tau'; \eta) \models_n^{int} \Gamma$ then for all $i \in 0..n-1$, we have $\delta; \tau'; \eta; i \vdash E \Downarrow G_i$ for some G_i .*

We now show the progress result for tables: every well-typed table with a conforming table-level input database evaluates in a well-formed evaluation environment to some output table:

Lemma 135 *If $\Gamma \vdash \mathbb{T} : Q$ and $\text{Core}(\mathbb{T})$ and $\mathbf{table}(Q)$ and $\text{fv}(Q) = \emptyset$ and $(\delta_{in}(t); \rho_{sz}) \models_{\rho_{sz}(t)}$ Q and $(\rho_{sz}; \delta; \tau'; \eta) \models_{\rho_{sz}(t)}^{int} \Gamma$ and $(\eta; \rho_{sz}) \models^{marg} \mathbb{T}$ then $t; \delta_{in}; \delta; \tau'; \eta \vdash \mathbb{T} \Downarrow \tau_{out}$ for some τ_{out} .*

Proof:

By induction on the derivation of $\Gamma \vdash \mathbb{T} : Q$:

- Case (TABLE CORE OUTPUT):

(TABLE CORE OUTPUT)

$$\frac{\Gamma \vdash^{\ell \wedge pc} E : T \quad \Gamma, x : \ell \wedge pc \vdash^{pc} \mathbb{T}' : Q' \quad c \notin \text{names}(Q)}{\Gamma \vdash^{pc} (c \triangleright x : T \text{ } \ell \text{ } \mathbf{output} \text{ } E) :: \mathbb{T}' : (c \triangleright x : T \text{ } (\ell \wedge pc) \text{ } \mathbf{output}) :: Q'}$$

– Subcase $\ell = \mathbf{static}$:

If $E = \mathbf{infer}.D[e_1, \dots, e_m].c_j(E')$, then $\Gamma \vdash^{\mathbf{static}} E : T$ must have been derived with (INFER):

$$\begin{array}{l} \text{(INFER) (where } \sigma(U) \triangleq U\{e_1/x_1\} \dots \{e_m/x_m\}) \\ D_{\mathbf{qry}} : [x_1 : T_1, \dots, x_m : T_m](c_1 : U_1, \dots, c_n : U_n) \rightarrow T' \\ \Gamma \vdash^{\mathbf{static}} e_i : T_i \quad \forall i \in 1..m \quad \Gamma \vdash^{\mathbf{static}} E' : \sigma(T') \quad j \in 1..n \\ \{x_1, \dots, x_m\} \cap (\bigcup_i \text{fv}(e_i)) = \emptyset \quad x_i \neq x_j \text{ for } i \neq j \\ \hline \Gamma \vdash^{\mathbf{static}} \mathbf{infer}.D[e_1, \dots, e_m].c_j(E') : \sigma(U_j) \end{array}$$

followed by 0 or more applications of (SUBSUM). Hence, we have $\Gamma \vdash \sigma(U_j) <: T$ (which in fact implies $\sigma(U_j) = T$, as U_j is in **qry**-space by inspection of the distribution signatures).

By Corollary 8, we have $\delta; \tau'; \eta; i \vdash e_i \Downarrow G'_i$ for some G'_i for all $i \in 1..m$.

By Corollary 7 and Lemma 128, $\emptyset \vdash^{\mathbf{static}} G'_i : T_i$. Moreover, since all T_i are scalar types and are in **det**-space, we have $G'_i = s_i$ for some s_i for all $i \in 1..m$.

By the derivation of $(\eta; \rho_{sz}) \models^{marg} \mathbb{T}$, we know that $\eta(c) = \mathbf{static}(v)$, where v is a measure on \mathcal{B} . Let:

$$(G_0, \dots, G_{\rho_{sz}(t)-1}) = \arg \min_{y_1, \dots, y_n} \|D[s_1, \dots, s_m](y_1, \dots, y_n) - v\|$$

(where $G_i = \text{fail}$ for all i if the $\arg \min$ does not exist).

As the $\arg \min$ operator takes a minimum over well-typed values (or returns a tuple of exceptions **fail**, which check against any type), we have

$$\emptyset \vdash^{\mathbf{static}} G_j : U_j \{s_1/x_1\} \dots \{s_m/x_m\}.$$

As $\sigma(U_j) = T$ is a top-level column type, we have $\text{fv}(T) = \emptyset$ by assumption. By Lemma 128, $\sigma(U_j)$ also contains no table size references. Thus, by Lemma 126, we have $\rho_{sz}(T) = \rho_{sz}(\sigma(U_j)) = U_j \{s_1/x_1\} \dots \{s_m/x_m\}$, so $\emptyset \vdash^{\mathbf{static}} G_j : \rho_{sz}(T)$.

By (CONF VAR STATIC), we have $(\rho_{sz}; \delta; \tau', (x \mapsto \mathbf{static}(G_j)); \eta) \models_{\rho_{sz}(t)}^{int} \Gamma, x : \mathbf{static} T$. By induction hypothesis, $t; \delta_{in}; \delta; \tau', (x \mapsto \mathbf{static}(G_j)); \eta \vdash \mathbb{T}' \Downarrow \tau'_{out}$ for some τ'_{out} . Thus, by (VAL QUERY STATIC),

$t; \delta_{in}; \delta; \tau'; \eta \vdash (c \triangleright x : T \text{ static output infer}.D[e_1, \dots, e_m].c_j(E')) :: \mathbb{T}' \Downarrow$
 $(c \mapsto \text{static}(G_j)) :: \tau'_{out}$.

If $E \neq \text{infer}.D[e_1, \dots, e_m].c_j(E')$ and $\neg \text{rnd}(T)$, then by Corollary 8, we get
 $\delta; \tau'; \eta; 0 \vdash E \Downarrow G$ for some G , where $\text{value}(G)$ is $\text{det}(T)$. By Corollary 7,
 $\emptyset \vdash^{\text{static}} G : \rho_{sz}(\tau'(T))$.

By (CONF VAR STATIC), $(\rho_{sz}; \delta; \tau', (x \mapsto \text{static}(G)); \eta) \models_{\rho_{sz}(t)}^{int} \Gamma, x : \text{static}$
 T . Hence, by induction hypothesis, we have $t; \delta_{in}; \delta; \tau', (x \mapsto \text{static}(G)); \eta \vdash$
 $\mathbb{T}' \Downarrow \tau'_{out}$ for some τ'_{out} . Thus, by (VAL QUERYORDET STATIC), $t; \delta_{in}; \delta; \tau'; \eta \vdash$
 $(c \triangleright x : T \text{ static output } E) :: \mathbb{T}' \Downarrow (c \mapsto \text{static}(G)) :: \tau'_{out}$.

If $\text{rnd}(T)$, then we have $(\rho_{sz}; \delta; \tau'; \eta) \models_{\rho_{sz}(t)}^{int} \Gamma, x : \text{static } T$ by (CONF VAR
 RND). Hence, $t; \delta_{in}; \delta; \tau'; \eta \vdash \mathbb{T}' \Downarrow \tau'_{out}$ follows by induction hypothesis,
 and so $t; \delta_{in}; \delta; \tau'; \eta \vdash (c \triangleright x : T \text{ static output } E) :: \mathbb{T}' \Downarrow \tau'_{out}$ by (VAL RAN-
 DOM).

– Subcase $\ell = \text{inst}$: similar (repeating the reasoning for every row).

• Case (TABLE CORE LOCAL): similar to (TABLE CORE OUTPUT).

• Case (TABLE INPUT):

(TABLE INPUT)

$$\frac{\Gamma, x : \ell \wedge pc \ T \vdash^{pc} \mathbb{T}' : Q' \quad c \notin \text{names}(Q')}{\Gamma \vdash^{pc} (c \triangleright x : T \ \ell \text{ input } \varepsilon) :: \mathbb{T}' : (c \triangleright x : T \ (\ell \wedge pc) \text{ input}) :: Q'}$$

– Subcase $\ell = \text{static}$:

Here, $(\delta_{in}(t), \rho_{sz}) \models_{\rho_{sz}(t)} (c \triangleright x : T \text{ static input}) :: Q'$ must have been derived
 by (CONF STATIC INPUT), so we have $\delta_{in}(t)(c) = \text{static}(V)$ (note that we
 have $\text{value}(V)$, as there are no exceptions in the database) and $\emptyset \vdash^{pc} V :$
 $\rho_{sz}(T)$ and $(\delta(t), \rho_{sz}) \models_n Q'$ and $\neg \text{det}(T)$.

First, suppose that $\neg \text{rnd}(T)$. Since $\text{fv}(T) = \emptyset$ by the **table** predicate,
 by (CONF VAR STATIC) we have $(\rho_{sz}; \delta; \tau', (x \mapsto \text{static}(V)); \eta) \models_{\rho_{sz}(t)}^{int} \Gamma, x : \text{static } T$.

By induction hypothesis, we have $t; \delta_{in}; \delta; \tau', (x \mapsto \text{static}(V)); \eta \vdash \mathbb{T}' \Downarrow \tau'_{out}$
 for some τ'_{out} .

Hence, by (VAL INPUT), $t; \delta_{in}; \delta; \tau'; \eta \vdash (c \triangleright x : T \text{ static input } \varepsilon) :: \mathbb{T}' \Downarrow$
 $(c \mapsto \delta_{in}(t)(c)) :: \tau'_{out}$.

Meanwhile, if $\mathbf{rnd}(T)$, then, again, $(\rho_{sz}; \delta; \tau'; \eta) \models_{\rho_{sz}(t)}^{int} \Gamma, x : \mathbf{static} T$ by (CONF VAR RND). Hence, $t; \delta_{in}; \delta; \tau'; \eta \vdash \mathbb{T}' \Downarrow \tau'_{out}$ by the induction hypothesis, so $t; \delta_{in}; \delta; \tau'; \eta \vdash (c \triangleright x : T \mathbf{static input} \varepsilon) :: \mathbb{T}' \Downarrow \tau'_{out}$ by (VAL RANDOM).

– Subcase $\ell = \mathbf{inst}$: similar

- Case (TABLE \square):

(TABLE \square)

$\Gamma \vdash \diamond$

$\Gamma \vdash^{pc} \square : \square$

Here, we get $t; \delta_{in}; \delta; \tau'; \eta \vdash \square \Downarrow \square$ immediately by (VAL EMPTY).

■

We finally obtain the desired progress result for schemas.

Lemma 136 *If $\Gamma \vdash \mathbb{S} : Sty$ and $\text{Core}(\mathbb{S})$ and $(\delta_{in}, \rho_{sz}) \models Sty$ and $(\rho_{sz}, \delta) \models^{int} \Gamma$ and $(\sigma, \rho_{sz}) \models^{marg} \mathbb{S}$ then $\delta_{in}, \delta \vdash \mathbb{S} \Downarrow_{\sigma} \delta_{out}$ for some δ_{out} .*

Proof: By straightforward induction on the derivation of $\Gamma \vdash \mathbb{S} : Sty$, with appeal to Lemma 135

■

Restatement of Lemma 13 *If $\emptyset \vdash \mathbb{S} : Sty$ and $\text{Core}(\mathbb{S})$ and $(\delta_{in}, \rho_{sz}) \models Sty$ and $(\sigma, \rho_{sz}) \models^{marg} \mathbb{S}$ then $\delta_{in}, \square \vdash \mathbb{S} \Downarrow_{\sigma} \delta_{out}$ for some δ_{out} .*

Proof: Corollary of Lemma 136.

■

Appendix D

Proof of Lemma 18 in Chapter 5

Restatement of Lemma 18 *If $\Gamma; \vec{e}; \vec{f}; \ell \wedge pc \vdash r ! \Pi$ and $K = \lambda E. (c \triangleright x : \mathbf{real} ! \mathbf{rnd}[\vec{e}] \ell \text{ viz } E) :: \mathbb{T}$ and $\Gamma, x : \ell \wedge pc \mathbf{real} ! \mathbf{rnd}[\vec{e}] \vdash^{pc} \mathbb{T} : Q$ and $\Gamma; \vec{e}; \ell \vdash v_i : \mathbf{mod}(f_i) ! \mathbf{det}$ for all $i \in 1..n$, then*

- *If $\text{viz} = \mathbf{output}$, then $\Gamma \vdash^{pc} \llbracket \ell; \vec{e}; \vec{f}; \vec{v}; r; K \rrbracket^\dagger : \llbracket \Pi \rrbracket @ ((c \triangleright x : \mathbf{real} ! \mathbf{rnd}[\vec{e}] (\ell \wedge pc) \mathbf{output})) :: Q$*
- *If $\text{viz} = \mathbf{local}$, then $\Gamma \vdash^{pc} \llbracket \ell; \vec{e}; \vec{f}; \vec{v}; r; K \rrbracket^\dagger : \llbracket \Pi \rrbracket @ Q$*

Proof: By induction on the derivation of $\Gamma; \vec{e}; \vec{f}; \ell \wedge pc \vdash r ! \Pi$, with appeal to Lemma 17:

- Case:

(NOISE) (where $\vec{e} = [e_1, \dots, e_n]$ and $\vec{f} = [f_1, \dots, f_{n'}]$ and $\sigma(U) \triangleq U \{\hat{e}_1/x_1\} \dots \{\hat{e}_{m'}/x_{m'}\}$)

$D_{rnd} : [x_1 : T_1, \dots, x_{m'} : T_{m'}](c_1 : U_1, \dots, c_m : U_m) \rightarrow \mathbf{real} ! \mathbf{rnd}$

$\Gamma \vdash \diamond \quad \Gamma \vdash^{\mathbf{static}} e_i : \mathbf{int} ! \mathbf{det} \quad \forall i \in 1..n \quad \Gamma \vdash^{\mathbf{static}} f_i : \mathbf{int} ! \mathbf{det} \quad \forall i \in 1..n'$

$\Gamma \vdash^{\mathbf{static}} \hat{e}_i : T_i \quad \forall i \in 1..m' \quad \Gamma; \vec{e}; \ell \wedge pc \vdash u_j : \sigma(U_j) \quad \forall j \in 1..m$

$\{x_1, \dots, x_{m'}\} \cap (\bigcup_i \text{fv}(\hat{e}_i)) = \emptyset \quad x_i \neq x_j \text{ for } i \neq j$

$\Gamma; \vec{e}; \vec{f}; \ell \wedge pc \vdash D[\hat{e}_1, \dots, \hat{e}_{m'}](u_1, \dots, u_m) ! \emptyset$

Here, $\llbracket \ell; \vec{e}; \vec{f}; \vec{v}; D[\hat{e}_1, \dots, \hat{e}_{m'}](u_1, \dots, u_m); K \rrbracket^\dagger = K [\mathbf{for} \vec{z} < \vec{e} \rightarrow D(\llbracket u_1 \rrbracket \vec{z}, \dots, \llbracket u_m \rrbracket \vec{z})] = (c \triangleright x : \mathbf{real} ! \mathbf{rnd}[\vec{e}] \ell \text{ viz } [\mathbf{for} \vec{z} < \vec{e} \rightarrow D[\hat{e}_1, \dots, \hat{e}_{m'}](\llbracket u_1 \rrbracket \vec{z}, \dots, \llbracket u_m \rrbracket \vec{z})]) :: \mathbb{T}$ and $\Pi = \emptyset$.

By Lemma 17 we have $\Gamma, \vec{z} : \ell \wedge pc \mathbf{mod}(\vec{e}) ! \mathbf{det} \vdash^{\ell \wedge pc} \llbracket u_j \rrbracket \vec{z} : \sigma(U_j)$ for all $j \in 1..m$.

Hence, by (RANDOM), we have

$\Gamma, \vec{z} : \ell \wedge pc \mathbf{mod}(\vec{e}) ! \mathbf{det} \vdash^{\ell \wedge pc} D[\hat{e}_1, \dots, \hat{e}_{m'}](\llbracket u_1 \rrbracket \vec{z}, \dots, \llbracket u_m \rrbracket \vec{z}) : \mathbf{real} ! \mathbf{rnd}$, and so

by (CUBE ITER), $\Gamma \vdash^{\ell \wedge pc} [\mathbf{for} \vec{z} < \vec{e} \rightarrow D[\hat{e}_1, \dots, \hat{e}_{m'}](\llbracket u_1 \rrbracket \vec{z}, \dots, \llbracket u_m \rrbracket \vec{z})] : (\mathbf{real} ! \mathbf{rnd})[\vec{e}]$. This implies $\Gamma \vdash^{\ell \wedge pc} [\mathbf{for} \vec{z} < \vec{e} \rightarrow D[\hat{e}_1, \dots, \hat{e}_{m'}](\llbracket u_1 \rrbracket \vec{z}, \dots, \llbracket u_m \rrbracket \vec{z})] : (\mathbf{real} ! \mathbf{rnd})[\vec{e}]$.

By assumption, $\Gamma, x :^{\ell \wedge pc} \mathbf{real} ! \mathbf{rnd}[\vec{e}] \vdash^{pc} \mathbb{T} : Q$.

– Subcase $viz = \mathbf{output}$:

By (TABLE CORE OUTPUT), we have $\Gamma \vdash^{pc} (c \triangleright x : \mathbf{real} ! \mathbf{rnd}[\vec{e}] \ell \mathbf{output} [\mathbf{for} \vec{z} < \vec{e} \rightarrow D[\hat{e}_1, \dots, \hat{e}_{m'}](\llbracket u_1 \rrbracket \vec{z}, \dots, \llbracket u_m \rrbracket \vec{z})]) :: \mathbb{T} : (c \triangleright x : \mathbf{real} ! \mathbf{rnd}[\vec{e}] (\ell \wedge pc) \mathbf{output}) :: Q$, as required.

– Subcase $viz = \mathbf{local}$:

By (TABLE CORE LOCAL), we have $\Gamma \vdash^{pc} (c \triangleright x : \mathbf{real} ! \mathbf{rnd}[\vec{e}] \ell \mathbf{local} [\mathbf{for} \vec{z} < \vec{e} \rightarrow D[\hat{e}_1, \dots, \hat{e}_{m'}](\llbracket u_1 \rrbracket \vec{z}, \dots, \llbracket u_m \rrbracket \vec{z})]) :: \mathbb{T} : Q$, as required.

• Case:

(COEFF)

$$\frac{\Gamma; \vec{e}; \ell \wedge pc \vdash v : \mathbf{real} ! \mathbf{rnd} \quad \Gamma; \vec{f}; []; \mathbf{static} \vdash r ! \Pi}{\Gamma; \vec{e}; \vec{f}; \ell \wedge pc \vdash v \{ \alpha \sim r \} ! \Pi, \alpha : (\mathbf{real} ! \mathbf{rnd})[\vec{f}]}$$

We have

$$\begin{aligned} & \llbracket \ell; \vec{e}; \vec{f}; \vec{v}; v \{ \alpha \sim r' \}; K \rrbracket^\dagger \\ = & \llbracket \mathbf{static}; \vec{f}; []; []; r'; \lambda E'. (\alpha \triangleright y : \mathbf{real} ! \mathbf{rnd}[\vec{f}] \mathbf{static output} E') :: (K \langle y; \vec{e}; \vec{v}; v \{ \alpha \sim r' \} \rangle) \rrbracket^\dagger \\ = & \llbracket \mathbf{static}; \vec{f}; []; []; r'; K' \rrbracket^\dagger \end{aligned}$$

where $K' = \lambda E'. (\alpha \triangleright y : \mathbf{real} ! \mathbf{rnd}[\vec{f}] \mathbf{static output} E') :: (c \triangleright x : \mathbf{real} ! \mathbf{rnd}[\vec{e}] \ell viz \langle y; \vec{e}; \vec{v}; v \{ \alpha \sim r' \} \rangle) :: \mathbb{T}$ and $\langle y; \vec{e}; \vec{v}; v \{ \alpha \sim r' \} \rangle = [\mathbf{for} \vec{z} < \vec{e} \rightarrow \llbracket v \rrbracket \vec{z} \times y[\llbracket v_n \rrbracket \vec{z}] \dots [\llbracket v_1 \rrbracket \vec{z}]]$.

By Lemma 17 and weakening, we have $\Gamma, y :^{\mathbf{static}} \mathbf{real} ! \mathbf{rnd}[\vec{f}], \vec{z} :^{\ell \wedge pc} \mathbf{mod}(\vec{e}) \vdash^{\ell \wedge pc} \llbracket v \rrbracket \vec{z} :^{\ell \wedge pc} \mathbf{real} ! \mathbf{rnd}$ and $\Gamma, y :^{\mathbf{static}} \mathbf{real} ! \mathbf{rnd}[\vec{f}] \vdash^{\ell \wedge pc} \llbracket v_i \rrbracket \vec{z} : \mathbf{mod}(f_i) ! \mathbf{rnd}$ for all $i \in 1..n$.

By (INDEX VAR), we also have $\Gamma, y :^{\mathbf{static}} \mathbf{real} ! \mathbf{rnd}[\vec{f}], \vec{z} :^{\ell \wedge pc} \mathbf{mod}(\vec{e}) \vdash^{\ell \wedge pc} y : \mathbf{real} ! \mathbf{rnd}[\vec{f}]$.

Hence, by (CUBE INDEX), $\Gamma, y :^{\mathbf{static}} \mathbf{real} ! \mathbf{rnd}[\vec{f}], \vec{z} :^{\ell \wedge pc} \mathbf{mod}(\vec{e}) \vdash^{\ell \wedge pc} y[\llbracket v_n \rrbracket \vec{z} x] \dots [\llbracket v_1 \rrbracket \vec{z} x] : \mathbf{real} ! \mathbf{rnd}$.

Therefore, by (PRIM),

$$\Gamma, y :^{\mathbf{static}} \mathbf{real} ! \mathbf{rnd}[\vec{f}], \vec{z} :^{\ell \wedge pc} \mathbf{mod}(\vec{e}) \vdash^{\ell \wedge pc} \llbracket v \rrbracket \vec{z} \times y[\llbracket v_n \rrbracket \vec{z}] \dots [\llbracket v_1 \rrbracket \vec{z}] : \mathbf{real} ! \mathbf{rnd},$$

and so by (CUBE ITER),

$$\Gamma, y :^{\mathbf{static}} \mathbf{real} ! \mathbf{rnd}[\vec{f}] \vdash^{\ell \wedge pc} [\mathbf{for} \vec{z} < \vec{e} \rightarrow \llbracket v \rrbracket \vec{z} \times y[\llbracket v_n \rrbracket \vec{z}] \dots [\llbracket v_1 \rrbracket \vec{z}]] : \mathbf{real} ! \mathbf{rnd}[\vec{e}].$$

By the assumption $\Gamma, x :^{\ell \wedge pc} \mathbf{real} ! \mathbf{rnd}[\vec{e}] \vdash^{pc} \mathbb{T} : Q$ and weakening, we have

$$\Gamma, y :^{\mathbf{static}} \mathbf{real} ! \mathbf{rnd}[\vec{f}], x :^{\ell \wedge pc} \mathbf{real} ! \mathbf{rnd}[\vec{e}] \vdash^{pc} \mathbb{T} : Q$$

– Subcase $viz = \mathbf{output}$:

By (TABLE CORE OUTPUT), we have $\Gamma, y : \mathbf{static} \mathbf{real} ! \mathbf{rnd}[\vec{f}] \vdash^{pc} (c \triangleright x : \mathbf{real} ! \mathbf{rnd}[\vec{e}] \ell \mathbf{output} \langle y; \vec{e}; \vec{v}; v\{\alpha \sim r'\} \rangle) :: \mathbb{T} : (c \triangleright x : \mathbf{real} ! \mathbf{rnd}[\vec{e}] \ell \wedge pc \mathbf{output}) :: Q$.

Since $K' = \lambda E. (\alpha \triangleright y : \mathbf{real} ! \mathbf{rnd}[\vec{f}] \mathbf{static output} E) ::$

$(c \triangleright x : \mathbf{real} ! \mathbf{rnd}[\vec{e}] \ell \mathbf{output} \langle y; \vec{e}; \vec{v}; v\{\alpha \sim r'\} \rangle) :: \mathbb{T}$, by induction hypothesis, we have $\Gamma \vdash^{pc} \llbracket \mathbf{static}; \vec{f}; []; []; r'; K' \rrbracket^\dagger :$

$\llbracket \Pi \rrbracket @ [(\alpha \triangleright y : \mathbf{real} ! \mathbf{rnd}[\vec{f}] \mathbf{static output})] @ ((c \triangleright x : \mathbf{real} ! \mathbf{rnd}[\vec{e}] \ell \wedge pc \mathbf{output}) :: Q)$, as required.

– Subcase $viz = \mathbf{local}$:

By (TABLE CORE LOCAL), we have $\Gamma, y : \mathbf{static} \mathbf{real} ! \mathbf{rnd}[\vec{f}] \vdash^{pc} (c \triangleright x : \mathbf{real} ! \mathbf{rnd}[\vec{e}] \ell \mathbf{local} \langle y; \vec{e}; \vec{v}; v\{\alpha \sim r'\} \rangle) :: \mathbb{T} : Q$.

Since $K' = \lambda E. (\alpha \triangleright y : \mathbf{real} ! \mathbf{rnd}[\vec{f}] \mathbf{static output} E) ::$

$(c \triangleright x : \mathbf{real} ! \mathbf{rnd}[\vec{e}] \ell \mathbf{local} \langle y; \vec{e}; \vec{v}; v\{\alpha \sim r'\} \rangle) :: \mathbb{T}$, by induction hypothesis,

we have $\Gamma \vdash^{pc} \llbracket \mathbf{static}; \vec{f}; []; []; r'; K' \rrbracket^\dagger : \llbracket \Pi \rrbracket @ [(\alpha \triangleright y : \mathbf{real} ! \mathbf{rnd}[\vec{f}] \mathbf{static output})] @ Q$, as required.

• Case:

(SUM)

$$\frac{\Gamma; \vec{e}; \vec{f}; \ell \wedge pc \vdash r_1 ! \Pi \quad \Gamma; \vec{e}; \vec{f}; \ell \wedge pc \vdash r_2 ! \Pi'}{\Gamma; \vec{e}; \vec{f}; \ell \wedge pc \vdash r_1 + r_2 ! \Pi, \Pi'}$$

In this case, $\llbracket \ell; \vec{e}; \vec{f}; \vec{v}; r_1 + r_2; K \rrbracket^\dagger =$

$\llbracket \ell; \vec{e}; \vec{f}; \vec{v}; r_1; \lambda E_1. (_ \triangleright y : \mathbf{real} ! \mathbf{rnd}[\vec{e}] \ell \mathbf{local} E_1) ::$

$\llbracket \ell; \vec{e}; \vec{f}; \vec{v}; r_2; \lambda E_2. (_ \triangleright z : \mathbf{real} ! \mathbf{rnd}[\vec{e}] \ell \mathbf{local} E_2) :: K [\mathbf{for} \vec{z} < \vec{e} \rightarrow y[\vec{z}] + z[\vec{z}]] \rrbracket^\dagger \rrbracket,$

where $y \notin \text{fv}(K) \cup \text{fv}(r_2)$ and $z \notin \text{fv}(K)$ and $K [\mathbf{for} \vec{z} < \vec{e} \rightarrow y[\vec{z}] + z[\vec{z}]] = (c \triangleright x : \mathbf{real} ! \mathbf{rnd}[\vec{e}] \ell \mathbf{viz} [\mathbf{for} \vec{z} < \vec{e} \rightarrow y[\vec{z}] + z[\vec{z}]] :: \mathbb{T}$.

By assumption, we have $\Gamma, x : \ell \wedge pc \mathbf{real} ! \mathbf{rnd}[\vec{e}] \vdash^{pc} \mathbb{T} : Q$, so by weakening, $\Gamma, y : \ell \wedge pc \mathbf{real} ! \mathbf{rnd}[\vec{e}], z : \ell \wedge pc \mathbf{real} ! \mathbf{rnd}[\vec{e}], x : \ell \wedge pc \mathbf{real} ! \mathbf{rnd}[\vec{e}] \vdash^{pc} \mathbb{T} : Q$.

By (CUBE INDEX) and (PRIM), $\Gamma, y : \ell \wedge pc \mathbf{real} ! \mathbf{rnd}[\vec{e}], z : \ell \wedge pc \mathbf{real} ! \mathbf{rnd}[\vec{e}], \vec{z} : \mathbf{mod}(\vec{e}) \vdash^{\ell \wedge pc} y[\vec{z}] + z[\vec{z}] : \mathbf{real} ! \mathbf{rnd}$, and so, by (CUBE ITER), $\Gamma, y : \ell \wedge pc \mathbf{real} ! \mathbf{rnd}[\vec{e}], z : \ell \wedge pc \mathbf{real} ! \mathbf{rnd}[\vec{e}] \vdash^{\ell \wedge pc} [\mathbf{for} \vec{z} < \vec{e} \rightarrow y[\vec{z}] + z[\vec{z}]] : \mathbf{real} ! \mathbf{rnd}[\vec{e}]$.

– Subcase $viz = \mathbf{output}$:

By (TABLE CORE OUTPUT), $\Gamma, y :^{\ell \wedge pc} \mathbf{real} ! \mathbf{rnd}[\vec{e}], z :^{\ell \wedge pc} \mathbf{real} ! \mathbf{rnd}[\vec{e}] \vdash^{pc}$
 $(c \triangleright x : \mathbf{real} ! \mathbf{rnd}[\vec{e}] \ell \mathbf{output} [\mathbf{for} \vec{z} < \vec{e} \rightarrow y[\vec{z}] + z[\vec{z}]] :: \mathbb{T} : (c \triangleright x : \mathbf{real} ! \mathbf{rnd}[\vec{e}] \ell \wedge pc \mathbf{output}) ::$
 Q .

By weakening, $\Gamma, y :^{\ell \wedge pc} \mathbf{real} ! \mathbf{rnd}[\vec{e}]; \vec{e}; \vec{f}; \ell \wedge pc \vdash r_2 ! \Pi$ and $\Gamma, y :^{\ell \wedge pc} \mathbf{real} !$
 $\mathbf{rnd}[\vec{e}]; \vec{e}; \ell \wedge pc \vdash v_i : \mathbf{mod}(f_i) ! \mathbf{rnd}$ for all $i \in 1..n$.

By induction hypothesis,

$\Gamma, y :^{\ell \wedge pc} \mathbf{real} ! \mathbf{rnd}[\vec{e}] \vdash^{pc} \llbracket \ell; \vec{e}; \vec{f}; \vec{v}; r_2; \lambda E_2. (_ \triangleright z : \mathbf{real} ! \mathbf{rnd}[\vec{e}] \ell \mathbf{local} E_2) ::$
 $(c \triangleright x : \mathbf{real} ! \mathbf{rnd}[\vec{e}] \ell \mathbf{output} [\mathbf{for} \vec{z} < \vec{e} \rightarrow y[\vec{z}] + z[\vec{z}]] :: \mathbb{T} \rrbracket^\dagger : \llbracket \Pi' \rrbracket @ ((c \triangleright x :$
 $\mathbf{real} ! \mathbf{rnd}[\vec{e}] \ell \wedge pc \mathbf{output})) :: Q$.

Hence, by applying the induction hypothesis again, we get

$\Gamma \vdash^{pc} \llbracket \ell; \vec{e}; \vec{f}; \vec{v}; r_1; \lambda E_1. (_ \triangleright y : \mathbf{real} ! \mathbf{rnd}[\vec{e}] \ell \mathbf{local} E_1) :: \llbracket \ell; \vec{e}; \vec{f}; \vec{v}; r_2; \lambda E_2. (_ \triangleright$
 $z : \mathbf{real} ! \mathbf{rnd}[\vec{e}] \ell \mathbf{local} E_2) :: (c \triangleright x : \mathbf{real} ! \mathbf{rnd}[\vec{e}] \ell \mathbf{output} [\mathbf{for} \vec{z} < \vec{e} \rightarrow y[\vec{z}] + z[\vec{z}]] ::$
 $\mathbb{T} \rrbracket^\dagger \rrbracket : \llbracket \Pi, \Pi' \rrbracket @ ((c \triangleright x : \mathbf{real} ! \mathbf{rnd}[\vec{e}] \ell \wedge pc \mathbf{output})) :: Q$, as required.

– Subcase $viz = \mathbf{local}$: similar.

- Case:

(GROUP)

$$\frac{\Gamma; \vec{e}; \ell \vdash v : \mathbf{mod}(f) \quad \Gamma; \vec{e}; (f :: \vec{f}); \ell \vdash r' ! \Pi}{\Gamma; \vec{e}; \vec{f}; \ell \vdash r' | v ! \Pi}$$

We have $\llbracket x; \vec{e}; \vec{f}; \vec{v}; r' | v; K \rrbracket^\dagger = \llbracket x; \vec{e}; f :: \vec{f}; v :: \vec{v}; r'; K \rrbracket^\dagger$.

By induction hypothesis, $\Gamma \vdash^{pc} \llbracket x; \vec{e}; f :: \vec{f}; v :: \vec{v}; r'; K \rrbracket^\dagger : \llbracket \Pi \rrbracket @ ((c \triangleright x : \mathbf{real} ! \mathbf{rnd}[\vec{e}] \ell \wedge pc viz) ::$
 $Q)$, if $viz = \mathbf{output}$ and $\Gamma \vdash^{pc} \llbracket x; \vec{e}; f :: \vec{f}; v :: \vec{v}; r'; K \rrbracket^\dagger : \llbracket \Pi \rrbracket @ Q$, if $viz = \mathbf{local}$, as
required.

- Case:

(RES)

$$\frac{\Gamma; \vec{f}; []; \mathbf{static} \vdash r_1 ! \Pi \quad \Gamma, y :^{\mathbf{static}} (\mathbf{real} ! \mathbf{rnd})[\vec{f}]; \vec{e}; \vec{f}; \ell \wedge pc \vdash r_2 ! \Pi'}{\Gamma; \vec{e}; \vec{f}; \ell \wedge pc \vdash y \sim r_1 \mathbf{in} r_2 ! \Pi, \Pi'}$$

Here, $\llbracket \ell; \vec{e}; \vec{f}; \vec{v}; y \sim r \mathbf{in} r'; K \rrbracket^\dagger = \llbracket \ell; \vec{e}; \vec{f}; \vec{v}; y \sim r \mathbf{in} r'; \lambda E. (c \triangleright x : \mathbf{real} ! \mathbf{rnd}[\vec{e}] \ell viz E) ::$
 $\mathbb{T} \rrbracket^\dagger = \llbracket \mathbf{static}; \vec{f}; []; []; r; \lambda E'. (_ \triangleright y : \mathbf{real} ! \mathbf{rnd}[\vec{f}] \mathbf{static} \mathbf{local} E') :: \llbracket \ell; \vec{e}; \vec{f}; \vec{v}; r'; \lambda E. (c \triangleright$
 $x : \mathbf{real} ! \mathbf{rnd}[\vec{e}] \ell viz E) :: \mathbb{T} \rrbracket^\dagger \rrbracket^\dagger$, where $y \notin \text{fv}(K)$.

By assumption and weakening, $\Gamma, y :^{\mathbf{static}} \mathbf{real} ! \mathbf{rnd}[\vec{f}], x :^{\ell \wedge pc} \mathbf{real} ! \mathbf{rnd}[\vec{e}] \vdash^{pc} \mathbb{T} :$
 Q and $\Gamma, y :^{\mathbf{static}} \mathbf{real} ! \mathbf{rnd}[\vec{f}]; \vec{e}; \ell \wedge pc \vdash v_i : \mathbf{mod}(f_i) ! \mathbf{rnd}$ for all $i \in 1..n$.

– Subcase $viz = \mathbf{output}$:

By induction hypothesis, $\Gamma, y : \mathbf{static} \mathbf{real} ! \mathbf{rnd}[\vec{f}] \vdash^{pc} \llbracket \ell; \vec{e}; \vec{f}; \vec{v}; r'; \lambda E. (c \triangleright x : \mathbf{real} ! \mathbf{rnd}[\vec{e}] \ell \mathbf{output} E) :: \mathbb{T} \rrbracket^\dagger : \llbracket \Pi' \rrbracket @((c \triangleright x : \mathbf{real} ! \mathbf{rnd}[\vec{e}] \ell \wedge pc \mathbf{output}) :: Q)$.

By applying the induction hypothesis again, we get

$\Gamma \vdash^{pc} \llbracket \mathbf{static}; \vec{f}; []; []; r; \lambda E'. (_ \triangleright y : \mathbf{real} ! \mathbf{rnd}[\vec{f}] \mathbf{static} \mathbf{local} E') :: \llbracket \ell; \vec{e}; \vec{f}; \vec{v}; r'; \lambda E. (c \triangleright x : \mathbf{real} ! \mathbf{rnd}[\vec{e}] \ell \mathbf{output} E) :: \mathbb{T} \rrbracket^\dagger \rrbracket^\dagger : \llbracket \Pi, \Pi' \rrbracket @((c \triangleright x : \mathbf{real} ! \mathbf{rnd}[\vec{e}] \ell \wedge pc \mathbf{output}) :: Q)$, as required.

– Subcase $viz = \mathbf{local}$: similar.

■

Appendix E

Proofs of Lemmas in Chapters 6 and 7

This appendix contains the proofs of Lemma 41 and proofs of measurability of \mathbf{P}_M , \mathbf{O}_M , $\mathbf{P}_M^\mathcal{V}$, peval and q , as well as a proof that \mathcal{Q} is a probability kernel.

Restatement of Lemma 41 *If $(M, 1, s) \rightarrow (M', w, [])$ and $M' \Downarrow_{w'}^{s'} G$, then $M \Downarrow_{w \cdot w'}^{s @ s'} G$.*

Proof: By induction on the structure of M .

If $M = E[\text{fail}]$ for some $E \neq []$, the result follows immediately by Lemma 39. Now, let us assume that $M \neq E[\text{fail}]$.

- Base case: $M = R$:
 - If $M = g(\vec{c})$ or $M = c \ V$ or $M = T$, then M reduces to a generalised value in 1 step, so the result holds trivially (by one of the evaluation rules).
 - Case $M = \text{if true then } M_2 \text{ else } M_3$: We have $(\text{if true then } M_2 \text{ else } M_3, 1, []) \rightarrow (M_2, 1, [])$. By assumption, $M_2 \Downarrow_{w'}^{s'} G$. Thus, the desired result holds by (EVAL IF TRUE).
 - Case $M = \text{if false then } M_2 \text{ else } M_3$: analogous to the previous case.
 - Case $M = (\lambda x. N_1) \ V$: We have $((\lambda x. N_1) \ V, 1, []) \rightarrow (N_1\{V/x\}, 1, [])$. Since $(\lambda x. N_1)$ and V are already values and $N_1\{V/x\} \Downarrow_{w'}^{s'} G$ by assumption, (EVAL APPL) yields $(\lambda x. N_1) \ V \Downarrow_{w'}^{s'} G$.
 - Case $M = D(\vec{c})$: $(M, 1, s) \rightarrow (M', w, [])$ must have been derived with (RED RANDOM) or (RED RANDOM FAIL). In the former case, $s = [c]$, $M' = c$, and $w = \text{pdf}_D(\vec{c}, c)$, where $c > 0$. The second assumption then takes the form $c \Downarrow_1^\emptyset c$, so the required result follows from (EVAL RANDOM). The (RED RANDOM FAIL) case is similar, with the result following from (EVAL RANDOM FAIL).
 - Case $M = \text{score}(c)$, $c \in (0, 1]$: $(M, 1, s) \rightarrow (M', w, [])$ must have been derived with (RED SCORE), so $M' = \text{true}$, $w = c$ and $s = []$. Thus, the result then

follows from (EVAL SCORE).

- Induction step: $M = E[R]$, $E \neq []$, $R \neq \text{fail}$:

- Case $E = (\lambda x.L) E'$: $M = (\lambda x.L) E'[R]$.

We have $((\lambda x.L) E'[R], 1, s) \rightarrow ((\lambda x.L) E'[N], w, [])$ for some N , so by Lemmas 26 and 27, $(E'[R], 1, s) \rightarrow (E'[N], w, [])$. By assumption, $(\lambda x.L) E'[N] \Downarrow_{w'}^{s'} G$.

- If $(\lambda x.L) E'[N] \Downarrow_{w'}^{s'} G$ was derived with (EVAL APPL), then $E'[N] \Downarrow_{w_1}^{s_1} V$ and $(\lambda x.L) V \Downarrow_{w_2}^{s_2} G$, where $w' = w_1 w_2$ and $s' = s_1 @ s_2$. By induction hypothesis, $E'[R] \Downarrow_{ww_1}^{s @ s_1} V$, so (EVAL APPL) gives $(\lambda x.L) E'[R] \Downarrow_{ww'}^{s @ s'} G$, as required.
- If $(\lambda x.L) E'[N] \Downarrow_{w'}^{s'} G$ was derived with (EVAL APPL RAISE3), then $G = \text{fail}$ and $E'[N] \Downarrow_{w'}^{s'} \text{fail}$. By induction hypothesis, $E'[R] \Downarrow_{ww'}^{s @ s'} \text{fail}$, so by (EVAL APPL RAISE3), $(\lambda x.L) E'[R] \Downarrow_{ww'}^{s @ s'} \text{fail}$

- Case $E = E' L$: $M = E'[R] L$:

We have $(E'[R] L, 1, s) \rightarrow (E'[N] L, w, [])$ for some N , so by lemmas 26 and 27, $(E'[R], 1, s) \rightarrow (E'[N], w, [])$. By assumption, $E'[N] L \Downarrow_{w'}^{s'} G$.

- If $E'[N] L \Downarrow_{w'}^{s'} G$ was derived with (EVAL APPL), then $E'[N] \Downarrow_{s_1}^{w_1} (\lambda x.N')$, $L \Downarrow_{s_2}^{w_2} V$ and $N' \{V/x\} \Downarrow_{s_3}^{w_3} G$, where $w' = w_1 w_2 w_3$ and $s' = s_1 @ s_2 @ s_3$. By induction hypothesis, $E'[R] \Downarrow_{ww_1}^{s @ s_1} (\lambda x.N')$, so (EVAL APPL) gives $E'[R] L \Downarrow_{ww'}^{s @ s'} G$, as required.
- If $E'[N] L \Downarrow_{w'}^{s'} G$ was derived with (EVAL APPL RAISE1), then $G = \text{fail}$ and $E'[N] \Downarrow_{w'}^{s'} \text{fail}$. By induction hypothesis, $E'[R] \Downarrow_{ww'}^{s @ s'} \text{fail}$, so by (EVAL APPL RAISE1), $E'[R] L \Downarrow_{ww'}^{s @ s'} \text{fail}$
- If $E'[N] L \Downarrow_{w'}^{s'} G$ was derived with (EVAL APPL RAISE3), then $E'[N] \Downarrow_{w_1}^{s_1} (\lambda x.N')$ and $L \Downarrow_{w_2}^{s_2} \text{fail}$, where $w' = w_1 w_2$ and $s' = s_1 @ s_2$. By induction hypothesis, $E'[R] \Downarrow_{ww_1}^{s @ s_1} (\lambda x.N')$, so (EVAL APPL RAISE3) gives $E'[R] L \Downarrow_{ww'}^{s @ s'} \text{fail}$, as required.
- If $E'[N] L \Downarrow_{w'}^{s'} G$ was derived with (EVAL APPL RAISE1), then $G = \text{fail}$ and $E'[N] \Downarrow_{w'}^{s'} c$. By induction hypothesis, $E'[R] \Downarrow_{ww'}^{s @ s'} c$, so by (EVAL APPL RAISE1), $E'[R] L \Downarrow_{ww'}^{s @ s'} \text{fail}$.

■

The proofs of measurability usually proceed by decomposing the functions into simpler operations. However, unlike Toronto [2014], we do not define these functions entirely in terms of general measurable operators, because the scope for reuse is limited here. We would have, for instance, to define multiple functions projecting different subexpressions of different expressions, and prove them measurable. Hence,

the overhead resulting from these extra definitions would be greater than the benefits.

A convenient way of showing that a function is Borel-measurable is to show that it is continuous as a function between metric spaces. If we have a function between products of metric spaces which is continuous with respect to the Manhattan product metric, then it is measurable with respect to the σ -algebras induced by the Manhattan metrics. We want to show that these σ -algebras are just products of σ -algebras induced by individual metrics. To this end, we can use the following standard result in measure theory:

Lemma 137 (Gallay [2009, Proposition 4.2 b]) *If (X_1, d_1) and (X_2, d_2) are separable metric spaces then*

$$\mathcal{B}(X_1 \times X_2) = \mathcal{B}(X_1) \times \mathcal{B}(X_2)$$

where $\mathcal{B}(X)$ is the σ -algebra induced by the metric space (X, d)

Functions in this chapter will be defined on subspaces of (Λ, \mathcal{M}) , $(\mathbb{R}, \mathcal{B})$ and $(\mathbb{U}, \mathcal{S})$, and their products. We already know that $(\Lambda, C\Lambda)$ is induced by the metric on terms defined in Section 6.3.1 and that $(\mathbb{U}, \mathcal{S})$ is induced by a metric on traces. Obviously, $(\mathbb{R}, \mathcal{B})$ is induced by the standard metric on \mathbb{R} . It is easy to show that (\mathbb{R}, d) and (\mathbb{U}, d) are separable. We can also show that this property holds for (Λ, d) :

Lemma 138 Λ_Q *is a dense subset of (Λ, d)*

Proof: We need to prove that

$$\forall M \in \Lambda, \varepsilon > 0 \exists M_Q \in \Lambda_Q \quad d(M, M_Q) < \varepsilon$$

This can be easily shown by induction (the base case follows from the fact that \mathbb{Q} is a dense subset of \mathbb{R}). ■

Corollary 9 *The metric space (Λ, d) is separable.*

It is clear that if a metric space (X, d) is separable, then for all $A \subseteq X$, the restricted space (A, d) is also separable, so, for instance, $(C\Lambda, d)$ is a separable metric space.

Corollary 10 *The σ -algebra on $C\Lambda \times \mathbb{R} \times \mathbb{U}$ induced by the Manhattan metric d defined as:*

$$d((M, w, s), (M', w', s')) \triangleq d(M, M') + d(w, w') + d(s, s')$$

is $\mathcal{M}|_{C\Lambda} \times \mathcal{B} \times \mathcal{S}$.

Note: Throughout this appendix, we call a function between metric spaces “measurable” if it is Borel measurable (with respect to σ -algebras induced by the metrics), unless stated otherwise.

The following lemma will also be useful in proving measurability of \mathbf{P}_M , \mathbf{O}_M and $\mathbf{P}_M^{\mathcal{V}}$.

Lemma 139 (Billingsley [1995, ex. 13.1]) *Let (Ω, Σ) and (Ω', Σ') be two measurable spaces, $T : \Omega \rightarrow \Omega'$ a function and A_1, A_2, \dots a countable collection of sets in Σ whose union is Ω . Let $\Sigma_n = \{A \mid A \subseteq A_n, A \in \Sigma\}$ be a σ -algebra in A_n and $T_n : A_n \rightarrow \Omega'$ a restriction of T to A_n . Then T is measurable Σ/Σ' if and only if T_n is measurable Σ_n/Σ' for every n .*

We can use Lemma 139 to split the space $C\Lambda$ of closed expressions into subspaces of expressions of different type, and restrict functions (such as the reduction relation) to a given type of expression, to process different cases separately.

We write $\mathbf{Subst}(M, x, v)$ for $M\{V/x\}$, to emphasize the fact that substitution is a function.

Detailed definition of substitution

$$\begin{aligned}
\mathbf{Subst}(c, x, V) &\triangleq c \\
\mathbf{Subst}(x, x, V) &\triangleq V \\
\mathbf{Subst}(x, y, V) &\triangleq y \quad \text{if } x \neq y \\
\mathbf{Subst}(\lambda x.M, x, V) &\triangleq \lambda x.M \\
\mathbf{Subst}(\lambda x.M, y, V) &\triangleq \lambda x.(\mathbf{Subst}(M, y, V)) \quad \text{if } x \neq y \\
\mathbf{Subst}(M \ N, x, V) &\triangleq \mathbf{Subst}(M, x, V) \ \mathbf{Subst}(N, x, V) \\
\mathbf{Subst}(D(V_1, \dots, V_{|D|}), x, V) &\triangleq D(\mathbf{Subst}(V_1, x, V), \dots, \mathbf{Subst}(V_{|D|}, x, V)) \\
\mathbf{Subst}(g(V_1, \dots, V_{|g|}), x, V) &\triangleq g(\mathbf{Subst}(V_1, x, V), \dots, \mathbf{Subst}(V_{|g|}, x, V)) \\
\mathbf{Subst}(\text{if } W \text{ then } M \text{ else } L, x, V) &\triangleq \\
&\quad \text{if } \mathbf{Subst}(W, x, V) \text{ then } \mathbf{Subst}(M, x, V) \text{ else } \mathbf{Subst}(L, x, V) \\
\mathbf{Subst}(\text{score}(V'), x, V) &\triangleq \text{score}(\mathbf{Subst}(V', x, V)) \\
\mathbf{Subst}(\text{fail}, x, V) &\triangleq \text{fail}
\end{aligned}$$

For convenience, let us also define a metric on contexts:

$$\begin{aligned}
d([\cdot], [\cdot]) &\triangleq 0 \\
d(EM, FN) &\triangleq d(E, F) + d(M, N) \\
d((\lambda x.M)E, (\lambda x.N)F) &\triangleq d(M, N) + d(E, F) \\
d(E, F) &\triangleq \infty \text{ otherwise}
\end{aligned}$$

Lemma 140 $d(E[M], F[N]) \leq d(E, F) + d(M, N)$.

Proof: By induction on the structure of E .

If $d(E, F) = \infty$, then the result is obvious, since $d(M', N') \leq \infty$ for all M', N' .

Now let us assume $d(E, F) \neq \infty$ and prove the result by simultaneous induction on the structure on E and F :

- Case $E = F = [\cdot]$: in this case, $E[M] = M$, $F[N] = N$, and $d(E, F) = 0$, so obviously $d(E[M], F[N]) = d(E, F) + d(M, N)$
- Case $E = E' L_1$, $F = F' L_2$:

We have $d(E[M], F[N]) = d(E'[M] L_1, F'[N] L_2) = d(E'[M], F'[N]) + d(L_1, L_2)$.

By induction hypothesis, $d(E'[M], F'[N]) \leq d(E', F') + d(M, N)$, so $d(E[M], F[N]) \leq d(E', F') + d(M, N) + d(L_1, L_2) = d(E, F) + d(M, N)$.

- Case $E = (\lambda x.L_1) E'$, $F = (\lambda x.L_2) F'$:

We have $d(E[M], F[N]) = d((\lambda x.L_1)(E'[M]), (\lambda x.L_2)(F'[N])) = d(\lambda x.L_1, \lambda x.L_2) + d(E'[M], F'[N])$. By induction hypothesis, $d(E'[M], F'[N]) \leq d(E', F') + d(M, N)$, so $d(E[M], F[N]) \leq d(E', F') + d(\lambda x.L_1, \lambda x.L_2) + d(M, N) = d(E, F) + d(M, N)$.

■

Lemma 141 If $d(E, F) = \infty$, then for all R_1, R_2 , $d(E[R_1], F[R_2]) = \infty$.

Proof: By induction on the structure of E :

- If $E = [\cdot]$, then $d(E, F) = \infty$ implies $F \neq [\cdot]$:
 - If $F = (\lambda x.M) F'$, then $d(E[R_1], F[R_2]) = d(R_1, (\lambda x.M) F'[R_2]) = \infty$, because R_1 is either not an application or of the form $V_1 V_2$, and $F'[R_2]$ is not a value.

- If $F = F' N$, then $d(E[R_1], F[R_2]) = d(R_1, F'[R_2] N) = \infty$, because R_1 is either not an application or of the form $V_1 V_2$, and $F'[R_2]$ is not a value.
- If $E = (\lambda x.M) E'$, then:
 - If $F = F' N$, then $d(E[R_1], F[R_2]) = d(\lambda x.M, F'[R_2]) + d(E'[R_1], N) = \infty$, because $d(\lambda x.M, F'[R_2]) = \infty$, as $F'[R_2]$ cannot be a lambda-abstraction.
 - If $F = (\lambda x.N) F'$, then $d(E, F) = \infty$ implies that either $d(M, N) = \infty$ or $d(E', F') = \infty$. We have $d(E[R_1], F[R_2]) = d(M, N) + d(E'[R_1], F'[R_2])$. If $d(M, N) = \infty$, then obviously $d(E[R_1], F[R_2]) = \infty$. Otherwise, by induction hypothesis, $d(E', F') = \infty$ gives $d(E'[R_1], F'[R_2]) = \infty$, and so $d(E[R_1], F[R_2]) = \infty$.
- If $E = E' M$ and $F = F' N$, then $d(E, F) = \infty$ implies that either $d(M, N) = \infty$ or $d(E', F') = \infty$. We have $d(E[R_1], F[R_2]) = d(M, N) + d(E'[R_1], F'[R_2])$, so $d(E'[R_1], F'[R_2]) = \infty$ follows like in the previous case.

The property also holds in all remaining cases by symmetry of d . ■

Lemma 142 $d(E[R_1], F[R_2]) = d(E, F) + d(R_1, R_2)$.

Proof: If $d(E, F) = \infty$, then $d(E[R_1], F[R_2]) = \infty$ by Lemma 141, otherwise the proof is the same as the proof of lemma 140, with inequality replaced by equality when applying the induction hypothesis. ■

Lemma 143 $d(\text{Subst}(M, x, V), \text{Subst}(N, x, W)) \leq d(M, N) + k \cdot d(V, W)$ where k is the max of the multiplicities of x in M and N

Proof: By simultaneous induction on the structure of M and N . ■

Let \mathcal{C} denote the set of contexts and \mathcal{G} the set of primitive functions. Let:

- $\Lambda_{\text{appl}} \triangleq \{E[(\lambda x.M)V] \mid E \in \mathcal{C}, (\lambda x.M) \in C\Lambda, V \in \mathcal{V}\}$
- $\Lambda_{\text{aplc}} \triangleq \{E[c V] \mid E \in \mathcal{C}, c \in \mathbb{R}, V \in \mathcal{V}\}$
- $\Lambda_{\text{iftrue}} \triangleq \{E[\text{if true then } M \text{ else } N] \mid E \in \mathcal{C}, M, N \in C\Lambda\}$

- $\Lambda_{iffalse} \triangleq \{E[\text{if } \mathbf{false} \text{ then } M \text{ else } N] \mid E \in \mathcal{C}, M, N \in C\Lambda\}$
- $\Lambda_{fail} \triangleq \{E[\text{fail}] \mid E \in \mathcal{C} \setminus \{[]\}\}$
- $\Lambda_{prim}(g) \triangleq \{E[g(\vec{c})] \mid E \in \mathcal{C}, \vec{c} \in \mathbb{R}^{|\mathcal{G}|}\}$
- $\Lambda_{prim} \triangleq \bigcup_{g \in \mathcal{G}} \Lambda_{prim}(g)$
- $A\Lambda_{if} \triangleq \{E[\text{if } G \text{ then } M \text{ else } N] \mid E \in \mathcal{C}, M, N \in C\Lambda, G \in \mathcal{GV}\}$
- $\Lambda_{dist}(D) \triangleq \{E[D(\vec{c})] \mid E \in \mathcal{C}, \vec{c} \in \mathbb{R}^{|D|}\}$
- $\Lambda_{dist} \triangleq \bigcup_{D \in \mathcal{D}} \Lambda_{dist}(D)$
- $A\Lambda_{prim} \triangleq \bigcup_{g \in \mathcal{G}} E[g(G_1, \dots, G_{|g|})] \mid E \in \mathcal{C}, G_1, \dots, G_{|g|} \in \mathcal{GV}\}$
- $A\Lambda_{dist} \triangleq \bigcup_{D \in \mathcal{D}} E[D(G_1, \dots, G_{|D|})] \mid E \in \mathcal{C}, G_1, \dots, G_{|D|} \in \mathcal{GV}\}$
- $A\Lambda_{scr} \triangleq \{E[\text{score}(c)] \mid E \in \mathcal{C}, c \in \mathbb{R}\}$
- $\Lambda_{scr} \triangleq \{E[\text{score}(c)] \mid E \in \mathcal{C}, c \in (0, 1]\}$

Lemma 144 *All the sets above are measurable.*

Proof: All these sets except for Λ_{scr} are closed, so they are obviously measurable. The set Λ_{scr} is not closed (for example, we can define a sequence of points in Λ_{scr} whose limit is $\text{score}(0) \notin \Lambda_{scr}$), but it is still measurable:

Define a function $i_{scr} : A\Lambda_{scr} \rightarrow \mathbb{R}$ by $i_{scr}(E[\text{score}(c)]) = c$. This function is continuous and so measurable. Since the interval $(0, 1]$ is a Borel subset of \mathbb{R} , $i_{scr}^{-1}((0, 1]) = \Lambda_{scr}$ is measurable. ■

Now, we need to define the set of erroneous redexes of all types.

- $R\Lambda_{if} \triangleq A\Lambda_{if} \setminus (\Lambda_{iftrue} \cup \Lambda_{iffalse})$
- $R\Lambda_{prim} \triangleq A\Lambda_{prim} \setminus \Lambda_{prim}$
- $R\Lambda_{dist} \triangleq A\Lambda_{dist} \setminus \Lambda_{dist}$
- $R\Lambda_{scr} \triangleq A\Lambda_{scr} \setminus \Lambda_{scr}$
- $\Lambda_{error} \triangleq R\Lambda_{if} \cup R\Lambda_{prim} \cup R\Lambda_{dist} \cup R\Lambda_{scr}$

Lemma 145 *The set Λ_{error} is measurable.*

Proof: It is constructed from measurable sets by operations preserving measurability. ■

Define:

$$\Lambda_{det} = \Lambda_{appl} \cup \Lambda_{cappl} \cup \Lambda_{iftrue} \cup \Lambda_{iffalse} \cup \Lambda_{fail} \cup \Lambda_{prim} \cup \Lambda_{error}$$

Lemma 146 Λ_{det} is measurable.

Proof: Λ_{det} is a union of measurable sets. ■

Lemma 147 \mathcal{GV} is measurable.

Proof: It is easy to see that \mathcal{GV} is precisely the union of sets of all closed expressions of the form c , $\lambda x.M$ and fail , so it is closed, and hence measurable. ■

Lemma 148 \mathcal{V} is measurable.

Proof: \mathcal{V} is the union of sets of all closed expressions of the form c and $\lambda x.M$, so it is closed, and hence measurable. ■

E.1 Deterministic reduction as a measurable function

Let us define a function performing one step of the reduction relation. This function has to be defined piecewise. Let us start with sub-functions reducing deterministic redexes of the given type.

$$\begin{aligned} g_{appl} &: \Lambda_{appl} \rightarrow C\Lambda \\ g_{appl}(E[(\lambda x.M) V]) &= E[\mathbf{Subst}(M, x, v)] \end{aligned}$$

Lemma 149 g_{appl} is measurable.

Proof: By Lemma 142, we have $d(E[(\lambda x.M)V], F[(\lambda x.N)W]) = d(E, F) + d(M, N) + d(V, W)$ and by Lemma 143, $d(E[\mathbf{Subst}(M, x, V)], F[\mathbf{Subst}(N, x, W)]) \leq d(E, F) + d(M, N) + k \cdot d(V, W)$, where k is the maximum of the multiplicities of x in M and N .

For any $\varepsilon > 0$, take $\delta = \frac{\varepsilon}{k+1}$. Then, if $d(E[(\lambda x.M)V], F[(\lambda x.N)W]) < \delta$, then

$$\begin{aligned}
d(E[\mathbf{Subst}(M, x, V)], F[\mathbf{Subst}(N, x, W)]) &\leq d(E, F) + d(M, N) + k \cdot d(V, W) \\
&\leq (k+1) \cdot (d(E, F) + d(M, N) + d(V, W)) \\
&= (k+1) \cdot d(E[(\lambda x.M)V], F[(\lambda x.N)W]) \\
&< \varepsilon
\end{aligned}$$

Thus, g_{appl} is continuous, and so measurable. ■

$$\begin{aligned}
g_{aplc} &: \Lambda_{aplc} \rightarrow C\Lambda \\
g_{aplc}(E[c \ M]) &= E[\mathbf{fail}]
\end{aligned}$$

Lemma 150 g_{aplc} is measurable.

Proof: It is easy to check that g_{aplc} is continuous. ■

$$\begin{aligned}
g_{prim} &: \Lambda_{prim} \rightarrow C\Lambda \\
g_{prim}(E[g(\vec{c})]) &= E[\sigma_g(\vec{c})]
\end{aligned}$$

Lemma 151 g_{prim} is measurable.

Proof: By assumption, every primitive function g is measurable. g_{prim} is a composition of a function splitting a context and a redex, g and a function combining a context with a redex, all of which are measurable. ■

$$\begin{aligned}
g_{iftrue} &: \Lambda_{iftrue} \rightarrow C\Lambda \\
g_{iftrue}(E[\mathbf{if \ true \ then \ } M_1 \mathbf{ \ else \ } M_2]) &= E[M_1] \\
g_{iffalse} &: \Lambda_{iffalse} \rightarrow C\Lambda \\
g_{iffalse}(E[\mathbf{if \ false \ then \ } M_1 \mathbf{ \ else \ } M_2]) &= E[M_2]
\end{aligned}$$

Lemma 152 g_{iftrue} and $g_{iffalse}$ are measurable.

Proof: We have $d(E[\mathbf{if \ true \ then \ } M_1 \mathbf{ \ else \ } N_1], F[\mathbf{if \ true \ then \ } M_2 \mathbf{ \ else \ } N_2]) = d(E, F) + d(M_1, M_2) + d(N_1, N_2) \geq d(E[M_1], F[M_2])$, so g_{iftrue} is continuous, and so measurable, and similarly for $g_{iffalse}$. ■

$$\begin{aligned}
g_{fail} &: \Lambda_{fail} \rightarrow C\Lambda \\
g_{fail}(E[\text{fail}]) &= \text{fail}
\end{aligned}$$

Lemma 153 g_{fail} is measurable.

Proof: Obvious, since it is a constant function. ■

$$\begin{aligned}
g_{error} &: \Lambda_{error} \rightarrow C\Lambda \\
g_{error}(E[T]) &= E[\text{fail}]
\end{aligned}$$

Lemma 154 g_{error} is measurable.

Proof: We have $d(E[T_1], F[T_2]) \geq d(E, F) = d(E[\text{fail}], F[\text{fail}])$, so g_{error} is continuous and hence measurable. ■

$$\begin{aligned}
g'_{det} &: \Lambda_{det} \rightarrow C\Lambda \\
g'_{det} &= g_{appl} \cup g_{applt} \cup g_{prim} \cup g_{iftrue} \cup g_{iffalse} \cup g_{fail} \cup g_{error}
\end{aligned}$$

Lemma 155 g'_{det} is measurable.

Proof: Follows directly from Lemma 139. ■

Lemma 156 $M \xrightarrow{det} N$ if and only if $g'_{det}(M) = N$.

Proof: By inspection. ■

E.2 Small- step reduction as a measurable function

Let

$$\begin{aligned}\mathcal{T}_{val} &= \mathcal{GV} \times \mathbb{R} \times \mathbb{U} \\ \mathcal{T}_{det} &= \Lambda_{det} \times \mathbb{R} \times \mathbb{U} \\ \mathcal{T}_{scr} &= \Lambda_{scr} \times \mathbb{R} \times \mathbb{U} \\ \mathcal{T}_{rnd} &= \{(E[D(\vec{c})], w, c :: s) \mid E \in \mathcal{C}, D \in \mathcal{D}, \vec{c} \in \mathbb{R}^{|\mathcal{D}|}, w \in \mathbb{R}, s \in \mathbb{U}, c \in \mathbb{R}, \\ &\quad \text{pdf}_D(\vec{c}, c) > 0\}\end{aligned}$$

Lemma 157 \mathcal{T}_{val} , \mathcal{T}_{det} , \mathcal{T}_{scr} and \mathcal{T}_{rnd} are measurable.

Proof: The measurability of \mathcal{T}_{val} , \mathcal{T}_{det} and \mathcal{T}_{scr} is obvious (they are products of measurable sets), so let us focus on \mathcal{T}_{rnd} .

For each distribution D , define a function $i_D : \Lambda_{rnd}(D) \times \mathbb{R} \times (\mathbb{U} \setminus \{\square\}) \rightarrow \mathbb{R}^{|\mathcal{D}|} \times \mathbb{R}$ by $i_D(E[D(\vec{c})], w, c :: s) = (c, \vec{c})$. This function is continuous, and so measurable. Then, since for each D , pdf_D is measurable by assumption, the function $j_D = \text{pdf}_D \circ i_D$ is measurable. Then, $\mathcal{T}_{rnd} = \bigcup_{D \in \mathcal{D}} j_D^{-1}((0, \infty))$, and since the set of distributions is countable, \mathcal{T}_{rnd} is measurable. ■

Let $\mathcal{T} = C\Lambda \times \mathbb{R} \times \mathbb{U}$ and let $\mathcal{T}_{blocked} = \mathcal{T} \setminus (\mathcal{T}_{val} \cup \mathcal{T}_{det} \cup \mathcal{T}_{scr} \cup \mathcal{T}_{rnd})$ be the set of non-reducible (“stuck”) triples, whose first components are not values. Obviously, $\mathcal{T}_{blocked}$ is measurable.

Define:

$$\begin{aligned}g_{val} &: \mathcal{T}_{val} \rightarrow \mathcal{T} \\ g_{val}(G, w, s) &= (\text{fail}, 0, \square)\end{aligned}$$

Obviously, g_{val} is measurable.

$$\begin{aligned}g_{det} &: \mathcal{T}_{det} \rightarrow \mathcal{T} \\ g_{det}(M, w, s) &= (g'_{det}(M), w, s)\end{aligned}$$

Lemma 158 g_{det} is measurable.

Proof: All components of g_{det} are measurable. ■

$$\begin{aligned}
g_{rnd} &: \mathcal{T}_{rnd} \rightarrow \mathcal{T} \\
g_{rnd} &\triangleq (g_1, g_2, g_3) \\
g_1(E[D(\vec{c})], w, c :: s) &\triangleq E[c] \\
g_2(E[D(\vec{c})], w, c :: s) &\triangleq w \cdot \text{pdf}_D(\vec{c}, c), \\
g_3(E[D(\vec{c})], w, c :: s) &\triangleq s
\end{aligned}$$

Lemma 159 g_{rnd} is measurable.

Proof: For g_1 , we have $d(E[c], E'[c']) \leq d(E, E') + d(c, c') \leq d(E, E') + d(\vec{c}, \vec{c}') + d(w, w') + d(s, s') = d((E[D(\vec{c})], w, c :: s), (E'[D(\vec{c}')] , w', c' :: s'))$ and $d((E[D(\vec{c})], w, c :: s), (E'[D(\vec{c}')] , w', c' :: s')) = \infty$ if $D \neq E$, so g_1 is continuous and hence Borel-measurable.

For g_2 , we have $g_2(E[D(\vec{c})], w, c :: s) = g_w(E[D(\vec{c})], w, c :: s) \times (\text{pdf}_D \circ g_c)(E[D(\vec{c})], w, c :: s)$, where $g_w(E[D(\vec{c})], w, c :: s) = w$ and $g_c(E[D(\vec{c})], w, c :: s) = (\vec{c}, c)$. The continuity (and so measurability) of g_w and g_c can be easily checked (as for g_1 above). Thus, $\text{pdf}_D \circ g_c$ is a composition of measurable functions (since distributions are assumed to be measurable), and so g_2 is a pointwise product of measurable real-valued functions, so it is measurable.

The continuity (and so measurability) of g_3 can be shown in a similar way to g_1 .

Hence, all the component functions of g_{rnd} are measurable, so g_{rnd} is itself measurable. ■

$$\begin{aligned}
g_{scr} &: \mathcal{T}_{scr} \rightarrow \mathcal{T} \\
g_{scr}(E[\text{score}(c)], w, s) &\triangleq (E[\text{true}], c \cdot w, s)
\end{aligned}$$

Lemma 160 g_{scr} is measurable.

Proof: The first component function of g_{scr} can easily be shown continuous, and so measurable, and ditto for the third component. The second component is a pointwise product of two measurable functions, like in the g_{rnd} case. Hence, g_{scr} is measurable. ■

For completeness, we also define:

$$\begin{aligned}
g_{\text{blocked}} &: \mathcal{T}_{\text{blocked}} \rightarrow \mathcal{T} \\
g_{\text{blocked}}(M, w, s) &\triangleq (\text{fail}, 0, [])
\end{aligned}$$

This function is trivially measurable.

Define

$$\begin{aligned}
g &: \mathcal{T} \rightarrow \mathcal{T} \\
g &\triangleq g_{\text{val}} \cup g_{\text{det}} \cup g_{\text{scr}} \cup g_{\text{blocked}}
\end{aligned}$$

Lemma 161 *g is measurable.*

Proof: Follows from Lemma 139. ■

Lemma 162 *For every $(M, w, s) \in \mathcal{T}$,*

- (1) *If $(M, w, s) \rightarrow (M', w', s')$, then $g(M, w, s) = (M', w', s')$.*
- (2) *If $g(M, w, s) = (M', w', s') \neq (\text{fail}, 0, [])$, then $(M, w, s) \rightarrow (M', w', s')$.*

Proof: By inspection. ■

E.3 Measurability of P and O

It is easy to check that the sets \mathcal{GV} and \mathbb{R}_+ (nonnegative reals) form $\omega\mathbf{CPO}$ s with the orderings $\text{fail} \leq M$ for all M and $0 \leq x$, respectively. This means that functions into \mathcal{GV} and \mathbb{R}_+ also form $\omega\mathbf{CPO}$ s with pointwise ordering.

Define:

$$\begin{aligned}
\Theta_{\Lambda}(f)(M, w, s) &\triangleq \begin{cases} M & \text{if } M \in \mathcal{GV}, s = [] \\ f(g(M, w, s)) & \text{otherwise} \end{cases} \\
\Theta_w(f)(M, w, s) &\triangleq \begin{cases} w & \text{if } M \in \mathcal{GV}, s = [] \\ f(g(M, w, s)) & \text{otherwise} \end{cases}
\end{aligned}$$

It can be shown that these functions are continuous, so we can define:

$$\perp_{\Lambda} = (M, w, s) \mapsto \text{fail}$$

$$\perp_w = (M, w, s) \mapsto 0$$

$$\mathbf{O}'(M, s) \triangleq \sup_n \Theta_\Lambda^n(\perp_\Lambda)(M, 1, s)$$

$$\mathbf{P}'(M, s) \triangleq \sup_n \Theta_w^n(\perp_w)(M, 1, s)$$

Lemma 163 *If $(M, w_0, s) \Rightarrow (G, w, \square)$, then $\sup_n \Theta_w^n(\perp_w)(M, w_0, s) = w$ and $\sup_n \Theta_\Lambda^n(\perp_\Lambda)(M, w_0, s) = G$.*

Proof: By induction on the derivation of $(M, w_0, s) \Rightarrow (G, w, \square)$:

- If $(M, w_0, s) \rightarrow^0 (G, w, \square)$, and so $M \in \mathcal{GV}$ and $s = \square$, then the equalities follow directly from the definitions of Θ_w and Θ_Λ .
- If $(M, w_0, s) \rightarrow (M', w', s') \Rightarrow (G, w, \square)$, assume that $\sup_n \Theta_w^n(\perp_w)(M', w', s') = w$ and $\sup_n \Theta_\Lambda^n(\perp_\Lambda)(M', w', s') = G$. We have $M \notin \mathcal{GV}$. By Lemma 162, $g(M, w_0, s) = (M', w', s')$. Hence

$$\begin{aligned} \sup_n \Theta_w^n(\perp_w)(M, w_0, s) &= \sup_n \Theta_w^n(\perp_w)(g(M, w_0, s)) \\ &= \sup_n \Theta_w^n(\perp_w)(M', w', s') = w \end{aligned}$$

by induction hypothesis. Similarly, $\sup_n \Theta_\Lambda^n(\perp_\Lambda)(M, w_0, s) = G$.

■

Corollary 11 *If $(M, 1, s) \Rightarrow (G, w, \square)$, then $\mathbf{P}'(M, s) = w$ and $\mathbf{O}'(M, s) = G$.*

Lemma 164 *If $\sup_n \Theta_w^n(\perp_w)(M, w_0, s) = w \neq 0$, then $(M, w_0, s) \Rightarrow (G, w, \square)$ for some $G \in \mathcal{GV}$.*

Proof: Because the supremum is taken with respect to a flat $\omega\mathbf{CPO}$, $\sup_n \Theta_w^n(\perp_w)(M, w_0, s) = w > 0$ implies $\Theta_w^k(\perp_w)(M, w_0, s) = w$ for some $k > 0$. We can then prove the result by induction on k :

- Base case, $k = 1$: We must have $\Theta_w(\perp_w)(M, w_0, s) = w_0$, $M = G \in \mathcal{GV}$ and $s = \square$ as otherwise we would obtain $\perp_w(M, w_0, s) = 0$. Hence (M, w_0, s) reduces to (G, w_0, \square) in 0 steps.

- Induction step: $\Theta_w^{k+1}(\perp_w)(M, w_0, s) = w$. If $M \in \mathcal{GV}$ and $s = []$, then $w = w_0$ and (M, w_0, s) reduces to itself in 0 steps, like in the base case. Otherwise, we have $\Theta_w^k(\perp_w)((M', w', s')) = w$, where $g(M, w_0, s) = (M', w', s')$. We know that $(M', w', s') \neq (\text{fail}, 0, [])$, because otherwise we would have $w = 0$. Thus, by Lemma 162, $(M, w_0, s) \rightarrow (M', w', s')$. By induction hypothesis, $(M', w', s') \Rightarrow (G, w, [])$, which implies $(M, w_0, s) \Rightarrow (G, w, [])$.

■

Lemma 165 *If $\sup_n \Theta_\Lambda^n(\perp_w)(M, w_0, s) = V \in \mathcal{V}$, then $(M, w_0, s) \Rightarrow (V, w, [])$ for some $w \in \mathbb{R}$.*

Proof: Similar to the proof of Lemma 164.

■

Corollary 12 *If there are no G, w such that $(M, 1, s) \Rightarrow (G, w, [])$, then $\mathbf{P}'(M, s) = 0$ and $\mathbf{O}'(M, s) = \text{fail}$.*

Corollary 13 *For any M , $\mathbf{P}_M = \mathbf{P}'(M, \cdot)$ and $\mathbf{O}_M = \mathbf{O}'(M, \cdot)$.*

Lemma 166 *If (X, Σ_1) and (Y, Σ_2) are measurable spaces, Y forms a flat $\omega\mathbf{CPO}$ with a bottom element \perp such that $\{\perp\} \in \Sigma_2$ and f_1, f_2, \dots is a ω -chain of Σ_1/Σ_2 measurable functions (on the $\omega\mathbf{CPO}$ with pointwise ordering), then $\sup_i f_i$ is Σ_1/Σ_2 measurable.*

Proof: Since $f^{-1}(A \cup \{\perp\}) = f^{-1}(A) \cup f^{-1}(\{\perp\})$, we only need to show that $(\sup_i f_i)^{-1}(\{\perp\}) \in \Sigma_1$ and $(\sup_i f_i)^{-1}(A) \in \Sigma_1$ for all $A \in \Sigma_2$ such that $\perp \notin A$.

We have $(\sup_i f_i)^{-1}(\{\perp\}) = \bigcap_i f_i^{-1}(\{\perp\})$, which is measurable by definition. If $\perp \notin A$, then $\sup_i f_i(x) \in A$ if and only if $f_i(x) \in A$ for some i , so by extensionality of sets, $\sup_i f_i^{-1}(A) = \bigcup_i f_i^{-1}(A) \subseteq \Sigma_1$.

■

Lemma 167 *\mathbf{P}' is measurable $(\mathcal{M}|_{C\Lambda} \times \mathcal{S})/\mathcal{R}|_{\mathbb{R}_+}$.*

Proof: First, let us show by induction on n that $\Theta_w^n(\perp_w)$ is measurable for every n :

- Base case, $n = 0$: $\Theta_w^0(\perp_w) = \perp_w$ is a constant function, and so trivially measurable.

- Induction step: suppose $\Theta_w^n(\perp_w)$ is measurable. Then we have $\Theta_w^{n+1}(\perp_w) = \Theta_w(\Theta_w^n(\perp_w))$, so it is enough to show that $\Theta_w(f)$ is measurable if f is measurable:

The domain of the first case is $\mathcal{GV} \times \mathbb{R} \times \{\emptyset\}$, which is clearly measurable. The domain of the second case is measurable as the complement of the above set in \mathcal{T} .

The sub-function corresponding to the first case returns the second component of its argument, so it is continuous and hence measurable. The second case is a composition of two measurable functions, hence measurable.

Thus, $\Theta_w(f)$ is measurable for any measurable f , and so $\Theta_w^{n+1}(\perp_w)$ is measurable.

By Lemma 166, $\sup_n \Theta_w^n(\perp_w)$ is measurable. Since \mathbf{P}' is a composition of $\sup_n \Theta_w^n(\perp_w)$ and a continuous function mapping (M, s) to $(M, 1, s)$, it is a composition of measurable functions, and so it is measurable. ■

Lemma 168 \mathbf{O}' is measurable $(\mathcal{M}|_{C\Lambda} \times \mathcal{S})/\mathcal{M}|_{\mathcal{GV}}$.

Proof: Similar to the proof of Lemma 167. ■

Restatement of Lemma 44 For any closed term M , the function \mathbf{P}_M is measurable $\mathcal{S}/\mathcal{R}|_{\mathbb{R}_+}$.

Proof: Since \mathbf{P}' is measurable, $\mathbf{P}_M = \mathbf{P}'(M, \cdot)$ is measurable for every $M \in C\Lambda$. ■

Restatement of Lemma 43 For each M , the function \mathbf{O}_M is measurable $\mathcal{S}/\mathcal{M}|_{\mathcal{GV}}$.

Proof: Since \mathbf{O}' is measurable, $\mathbf{O}_M = \mathbf{O}'(M, \cdot)$ is measurable for every $M \in C\Lambda$. ■

Lemma 169 For all M, s , $\mathbf{P}_M^\mathcal{V}(s) = \mathbf{P}_M(s)[\mathbf{O}_M(s) \in \mathcal{V}]$

Proof: By Lemma 42, if $M \Downarrow_s^w G$, then w, G are unique. If $M \Downarrow_s^w V$, then $\mathbf{P}_M(s) = w$, $\mathbf{P}_M^\mathcal{V}(s)$ and $\mathbf{O}_M(s) \in \mathcal{V}$, so the equality holds. If $M \Downarrow_s^w \text{fail}$, then $\mathbf{P}_M(s) = w$, $\mathbf{P}_M^\mathcal{V}(s) = 0$ and $\mathbf{O}_M(s) \notin \mathcal{V}$, so both sides of the equation are 0. If there is no G such that $M \Downarrow_s^w G$, then both sides are also 0. ■

Restatement of Lemma 46 $\mathbf{P}_M^{\mathcal{V}}$ is measurable \mathcal{S}/\mathbb{R}_+ for every M .

Proof: By Lemma 169, $\mathbf{P}_M^{\mathcal{V}}(s) = \mathbf{P}_M(s)[\mathbf{O}_M(s) \in \mathcal{V}]$, so $\mathbf{P}_M^{\mathcal{V}}$ is a pointwise product of a measurable function and a composition of \mathbf{O}_M and an indicator function for a measurable set, hence it is measurable. ■

To simplify the notation, let us write $R^n(M, w, s)$ for $\Phi^n(\perp_w)(M, w, s)$ in the subsequent lemmas.

Lemma 170 For every $M \in C\Lambda$, $n \geq 1$, $w \geq 0$ and s , $R^n(M, w, s) = wR^n(M, 1, s)$.

Proof: By induction on n :

- Base case: $n = 1$

We have $R(M, w, s) = w$ if $M \in \mathcal{GV}$ and $s = []$ and $R(M, w, s) = 0$ otherwise. In the former case, we also have $R(M, 1, s) = 1$, so obviously $wR(M, 1, s) = w = R(M, w, s)$. In the latter case, $R(M, 1, s) = 0$, so also $wR(M, 1, s) = 0 = R(M, w, s)$.

- Induction step:

Suppose the hypothesis holds for some n . By definition of R^n , we have:

$$R^{n+1}(M, w, s) = \begin{cases} w & \text{if } M \in \mathcal{GV}, s = [] \\ R^n(g(M, w, s)) & \text{otherwise} \end{cases}$$

If $M \in \mathcal{GV}$, $s = []$, then we have $R^{n+1}(M, 1, s) = 1$, so $wR^{n+1}(M, 1, s) = w = R^{n+1}(M, w, s)$.

Otherwise, we have $R^{n+1}(M, 1, s) = R^n(g(M, 1, s))$ and $g(M, w, s) = (M', w', s')$ for some M' , $w' \geq 0$ and s' .

If $w > 0$, then $g(M, 1, s) = (M', w'/w, s')$ by Lemma 29 and Lemma 162 (it is easy to check that if $(M', w', s') = (\text{fail}, 0, [])$, then $g(M, 1, s) = (\text{fail}, 0, [])$, as success or failure of reduction does not depend on the initial weight). By induction hypothesis, $R^n(M', w'/w, s') = (w'/w)R^n(M', 1, s')$ and $R^n(M', w', s') = w'R^n(M', 1, s')$. Hence, we have $R^{n+1}(M, w, s) = R^n(M', w', s') = w'R^n(M', 1, s')$ and $wR^{n+1}(M, 1, s) = wR^n(M', w'/w, s') = w'R^n(M', 1, s')$, so $R^{n+1}(M, w, s) = w'R^n(M', 1, s')$, as required.

Meanwhile, if $w = 0$, then obviously $w' = 0$, so $R^{n+1}(M, w, s) = R^n(M', 0, s') = 0$ by induction hypothesis.

■

Corollary 14 For every $M \in C\Lambda$, $n \geq 1$, $R^n(M, 0, s) = 0$.

Lemma 171 For any $M \in C\Lambda$ and $n \geq 1$, $\int R^n(M, 1, s) ds \leq 1$.

Proof: By induction on n :

- Base case: $n = 1$: $\int R^1(M, 1, s) ds \leq 1$. We have:

$$R^1(M, 1, s) = \begin{cases} 1 & \text{if } M \in \mathcal{GV}, s = []. \\ 0 & \text{otherwise} \end{cases}$$

Thus: $\int R^1(M, 1, s) ds = \int_{\{[]\}} R^1(M, 1, s) ds = R^1(M, 1, []) = [M \in \mathcal{GV}]$. This immediately gives $\int R^1(M, 1, s) ds \leq 1$.

- Induction step: Suppose $\int R^n(N, 1, s) ds \leq 1$ for every closed N and some $n \geq 1$. We need to show $\int R^{n+1}(M, 1, s) ds \leq 1$ for every closed M . We must consider several cases:

– Case $M = E[D(\vec{c})]$:

We have $g(E[D(\vec{c})], 1, []) = (\text{fail}, 0, [])$ and
 $g(E[D(\vec{c})], 1, c :: s) = (E[c], \text{pdf}_D(\vec{c}, c), s)$ if $\text{pdf}_D(\vec{c}, c) > 0$ and $g(E[D(\vec{c})], 1, c :: s) = (E[\text{fail}], 0, s)$ if $\text{pdf}_D(\vec{c}, c) = 0$.

We have $R^{n+1}(E[D(\vec{c})], 1, []) = R^n(E[\text{fail}], 0, []) = 0$ by Corollary 14. Meanwhile, by Lemma 170, $R^n(E[c], \text{pdf}_D(\vec{c}, c), s) = \text{pdf}_D(\vec{c}, c) R^n(E[c], 1, s)$ and by Corollary 14, $R^n(E[\text{fail}], 0, s) = R^n(E[c], 0, s) = 0$, so $R^{n+1}(E[D(\vec{c})], 1, c :: s) = \text{pdf}_D(\vec{c}, c) R^n(E[c], 1, s)$ for all $c \in \mathbb{R}$.

Hence, we have:

$$\begin{aligned}
& \int R^{n+1}(E[D(\vec{c})], 1, s) \mu(ds) \\
& \int_{\mathbb{U} \setminus \{\emptyset\}} R^{n+1}(E[D(\vec{c})], 1, s) \mu(ds) \\
&= \sum_{i=1}^{\infty} \int_{\mathbb{R}^i} R^{n+1}(E[D(\vec{c})], 1, s) \mu(ds) \\
& \quad \text{(by Thm 16.9 from [Billingsley, 1995])} \\
&= \sum_{i=1}^{\infty} \int_{\mathbb{R}^i} R^{n+1}(E[D(\vec{c})], 1, s) \lambda^i(ds) \\
& \quad \text{(by Lemma 187)} \\
&= \sum_{i=1}^{\infty} \int_{\mathbb{R}^i} \text{pdf}_D(\vec{c}, c) R^n(E[c], 1, s_{2..i}) \lambda^i(ds) \\
&= \sum_{i=1}^{\infty} \int_{\mathbb{R}} \text{pdf}_D(\vec{c}, c) \int_{\mathbb{R}^{i-1}} R^n(E[c], 1, s') \lambda^{i-1}(ds') \lambda(dc) \\
& \quad \text{(by Fubini's theorem)} \\
&= \int_{\mathbb{R}} \text{pdf}_D(\vec{c}, c) \sum_{i=0}^{\infty} \int_{\mathbb{R}^i} R^n(E[c], 1, s') \lambda^i(ds') \lambda(dc) \\
& \quad \text{(by Lemma 185)} \\
&= \int_{\mathbb{R}} \text{pdf}_D(\vec{c}, c) \left(\int R^n(E[c], 1, s') \mu(ds') \right) \lambda(dc) \\
&\leq \int_{\mathbb{R}} \text{pdf}_D(\vec{c}, c) \lambda(dc) \\
& \quad \text{(by induction hypothesis)} \\
&\leq 1
\end{aligned}$$

where the last inequality follows from the assumption that each distribution D is a subprobability measure with density pdf_D .

– Case $M = E[\text{score}(c)]$, $c \in (0, 1]$:

In this case, $g(E[\text{score}(c)], 1, s) = g(E[\text{true}], c, s)$, and so for any s , we have $R^{n+1}(E[\text{score}(c)], 1, s) = R^n(E[\text{true}], c, s) = cR^n(E[\text{true}], 1, s)$ by Lemma 170. Hence $\int R^{n+1}(E[\text{score}(c)], 1, s) ds = \int cR^n(E[\text{true}], 1, s) ds = c \int R^n(E[\text{true}], 1, s) ds \leq c$ by induction hypothesis.

Thus, $\int R^{n+1}(E[\text{score}(c)], 1, s) ds \leq 1$, as required.

– Case $M \in \Lambda_{det}$:

We have $g(M, 1, s) = (N, 1, s)$ for some fixed N for every s , and so $R^{n+1}(M, 1, s) = R^n(N, 1, s)$. Thus $\int R^{n+1}(M, 1, s) ds = \int R^n(N, 1, s) ds \leq 1$ by induction hypothesis.

- Case $M \in \mathcal{GV}$: Here, $g(M, 1, s) = (\text{fail}, 0, \square)$ for every s . We have $R^{n+1}(M, 1, \square) = 1$ and $R^{n+1}(M, 1, s) = R^n(\text{fail}, 0, \square) = 0$ if $s \neq \square$. Thus: $\int R^{n+1}(M, 1, s) ds = R^{n+1}(M, 1, \square) = 1$.

■

Lemma 172 *For every closed M and s , $\sup_n R^n(M, 1, s) = \lim_{n \rightarrow \infty} R^n(M, 1, s)$, where the supremum is taken with respect to the flat ωCPO on reals.*

Proof:

If $\sup_n R^n(M, 1, s) = w > 0$, then, since the supremum is taken with respect to a flat ωCPO , we must have $w = R^k(M, 1, s)$ for some k . It is easy to check that $R^l(M, 1, s) = R^k(M, 1, s) = w$ for all $l \geq k$, so $\lim_{n \rightarrow \infty} R^n(M, 1, s) = w$.

If $\sup_n R^n(M, 1, s) = 0$, then we must have $R^k(M, 1, s) = 0$ for all k , so $\lim_{n \rightarrow \infty} R^n(M, 1, s) = 0$.

■

Lemma 173 *For every closed M , $\int \mathbf{P}_M(s) ds \leq 1$*

Proof: For every s , we have $\mathbf{P}_M(s) = \sup_n R^n(M, 1, s) = \lim_{n \rightarrow \infty} R^n(M, 1, s)$ by Lemma 172. The sequence of function R^n is obviously pointwise non-decreasing. Hence, $\int \mathbf{P}_M(s) ds = \lim_{n \rightarrow \infty} \int R^n(M, 1, s) ds$ by the monotone convergence theorem. This implies $\int \mathbf{P}_M(s) ds \leq 1$ by Lemma 171.

■

Lemma 174 *For every closed M , $\int \mathbf{P}_M^\mathcal{V}(s) ds \leq 1$*

Proof: Obviously, $\mathbf{P}_M^\mathcal{V}(s) \leq \mathbf{P}_M(s)$ for every s , so $\int \mathbf{P}_M^\mathcal{V}(s) ds \leq \int \mathbf{P}_M(s) ds \leq 1$.

■

Restatement of Lemma 45 $\langle\langle M \rangle\rangle$ is a subprobability measure on $(\mathbb{U}, \mathcal{S})$.

Proof: Since $\mathbf{P}_M(s)$ is nonnegative for every s , the function $\langle\langle M \rangle\rangle$ is a measure of density \mathbf{P}_M with respect to the stock measure μ [Gallay, 2009, Section 2.3.3]. By Lemma 173, it is a subprobability measure.

■

Restatement of Theorem 5 $\llbracket M \rrbracket_{\mathbb{U}}$ is a subprobability measure on $(\mathcal{GV}, \mathcal{M}|_{\mathcal{GV}})$.

Proof: The function $\llbracket M \rrbracket_{\mathbb{U}}$ is a transformation of the measure $\langle\langle M \rangle\rangle$ on $(\mathbb{U}, \mathcal{S})$ by the $\mathcal{S}/\mathcal{M}|_{\mathcal{G}\mathcal{V}}$ -measurable function \mathbf{O}_M , so it is a measure on $(\mathcal{G}\mathcal{V}, \mathcal{M}|_{\mathcal{G}\mathcal{V}})$ [Billingsley, 1995, Section 13, Transformations of Measures]. Since $\langle\langle M \rangle\rangle(\mathbb{U}) \leq 1$ by Lemma 45, $\llbracket M \rrbracket_{\mathbb{U}}(\mathcal{G}\mathcal{V}) = \langle\langle M \rangle\rangle(\mathbf{O}_M^{-1}(\mathcal{G}\mathcal{V})) = \langle\langle M \rangle\rangle(\mathbb{U}) \leq 1$. ■

Restatement of Lemma 47 *For every closed M , $\langle\langle M \rangle\rangle^{\mathcal{V}}$ is a subprobability measure on $(\mathbb{U}, \mathcal{S})$*

Proof: As $\mathbf{P}_M^{\mathcal{V}}(s)$ is nonnegative for every s , $\langle\langle M \rangle\rangle^{\mathcal{V}}$ is a measure of density $\mathbf{P}_M^{\mathcal{V}}$ with respect to the stock measure μ . By Lemma 174, it is a subprobability measure. ■

Restatement of Lemma 48 *For every closed M , $\llbracket M \rrbracket_{\mathbb{U}}|_{\mathcal{V}}$ is a subprobability measure on $(\mathcal{G}\mathcal{V}, \mathcal{M}|_{\mathcal{G}\mathcal{V}})$*

Proof: Since $\llbracket M \rrbracket_{\mathbb{U}}|_{\mathcal{V}}(A) = \langle\langle M \rangle\rangle^{\mathcal{V}}(\mathbf{O}_M^{-1}(A))$, $\llbracket M \rrbracket_{\mathbb{U}}|_{\mathcal{V}}(A)$ is a transformation of the measure $\langle\langle M \rangle\rangle^{\mathcal{V}}$ by the $\mathcal{S}/\mathcal{M}|_{\mathcal{G}\mathcal{V}}$ -measurable function \mathbf{O}_M , so it is a measure on $(\mathcal{G}\mathcal{V}, \mathcal{M}|_{\mathcal{G}\mathcal{V}})$. We have $\llbracket M \rrbracket_{\mathbb{U}}|_{\mathcal{V}}(\mathcal{G}\mathcal{V}) = \langle\langle M \rangle\rangle^{\mathcal{V}}(\mathbf{O}_M^{-1}(\mathcal{G}\mathcal{V})) = \langle\langle M \rangle\rangle^{\mathcal{V}}(\mathbb{U}) \leq 1$, by Lemma 47, so $\llbracket M \rrbracket_{\mathbb{U}}|_{\mathcal{V}}$ is a subprobability measure. ■

E.4 Measurability of peval

Like in the previous section, we start by giving an alternative definition of `peval`, using the function `g` instead of referring to the reduction relation directly.

The set of closed terms CA is a $\omega\mathbf{CPO}$ with respect to the partial order defined by $\text{fail} \leq G$ for all G . Hence the set \mathbf{F} of all functions $(CA \times \mathbb{R} \times \mathbb{U}) \rightarrow CA$ is a $\omega\mathbf{CPO}$ with respect to the pointwise order. Define $\Phi : \mathbf{F} \rightarrow \mathbf{F}$ as:

$$\Phi(f)(M, w, s) = \begin{cases} M & \text{if } s = [] \\ f(g(M, w, s)) & \text{otherwise} \end{cases}$$

It is easy to check that Φ is monotone and preserves suprema of ω -chains, so it is continuous. Hence, we can define:

$$\text{peval}'(M, s) = \sup_k \Phi^k(\perp_{\Lambda})(M, 1, s)$$

where $\perp_{\Lambda}(M, w, s) = \text{fail}$, as before.

We first need to show that the original `peval` function is well-defined.

Lemma 175 *If $(M, w_0, s) \Rightarrow (M_k, w_k, s_k) \rightarrow (M', w', [])$ and $s_k \neq []$ and $(M, w_0, s) \Rightarrow (M_l, w_l, s_l) \rightarrow (M'', w'', [])$ and $s_l \neq []$, then $M' = M''$ and $w' = w''$.*

Proof: By induction on the derivation of $(M, w_0, s) \Rightarrow (M_k, w_k, s_k)$:

- If $(M, w_0, s) \Rightarrow (M_k, w_k, s_k)$ was derived in 0 steps, we have $M_k = M$, $w_k = w$ and $s_k = s$, and so $(M, w_0, s) \rightarrow (M', w', [])$, where $s \neq []$.

If $(M, w_0, s) \Rightarrow (M_l, w_l, s_l)$ was derived in 0 steps, then $(M_l, w_l, s_l) = (M, w_0, s)$, and so $M'' = M'$ and $w'' = w'$ by Lemma 25.

If $(M, w_0, s) \Rightarrow (M_l, w_l, s_l)$ was derived in 1 or more steps, we have $(M, w_0, s) \rightarrow (\hat{M}, \hat{w}, \hat{s}) \Rightarrow (M_l, w_l, s_l) \rightarrow (M'', w'', [])$ and $s_l \neq []$, for some $\hat{M}, \hat{w}, \hat{s}$. By Lemma 25, $\hat{s} = []$. We have $(\hat{M}, \hat{w}, []) \Rightarrow (M_l, w_l, s_l)$, where $s_l \neq []$. This leads to a contradiction, as it is easy to show that reducing a term with an empty trace cannot yield a triple with a non-empty trace (there is no rule which adds an element to a trace)

- If $(M, w_0, s) \Rightarrow (M_k, w_k, s_k)$ was derived in 1 or more steps, we have $(M, w_0, s) \rightarrow (M^*, w^*, s^*) \rightarrow^k (M_k, w_k, s_k) \rightarrow (M', w', [])$ for some $k \geq 0$, M^*, w^*, s^* . Now, if $(M, w_0, s) \Rightarrow (M_l, w_l, s_l)$ was derived in 1 or more steps, we have $(M, w_0, s) \rightarrow (\hat{M}, \hat{w}, \hat{s}) \Rightarrow (M_l, w_l, s_l) \rightarrow (M'', w'', [])$ and $s_l \neq []$ for some $\hat{M}, \hat{w}, \hat{s}$, where $(\hat{M}, \hat{w}, \hat{s}) = (M^*, w^*, s^*)$ by Lemma 25. Hence, the result follows by the induction hypothesis.

If $(M, w_0, s) \Rightarrow (M_l, w_l, s_l)$ was derived in 0 steps, then $(M_l, w_l, s_l) = (M, w_0, s)$, and so $(M, w_0, s) \rightarrow (M'', w'', [])$. By Lemma 25, this implies $s^* = []$, so $(M^*, w^*, []) \Rightarrow (M_k, w_k, s_k)$ for $s_k \neq []$, which is impossible, as explained in the previous case.

■

Lemma 176 *If $(M, w_0, s) \Rightarrow (M_k, w_k, s_k) \rightarrow (M', w', [])$ and $s_k \neq []$, then $\sup_n \Phi^n(\perp_{\Lambda})(M, w_0, s) = M'$.*

Proof: By induction on the length of derivation of $(M, w_0, s) \Rightarrow (M_k, w_k, s_k) \rightarrow (M', w', [])$.

Suppose $(M, w_0, s) \rightarrow^k (M_k, w_k, s_k) \rightarrow (M', w', [])$.

- Base case, $k = 0$: We have $(M, w_0, s) \rightarrow (M', w', \perp)$ and $s \neq \perp$. Hence, by Lemma 162, $g(M, w_0, s) = (M', w', \perp)$, and so, by monotonicity of Φ , $\sup_k \Phi^k(\perp_\Lambda)(M, w_0, s) = \sup_k \Phi(\Phi^k(\perp_\Lambda))(M, w_0, s) = \sup_k \Phi(\Phi^k(\perp_\Lambda))(M', w', \perp) = M'$, as required.
- Induction step: Let $(M, w_0, s) \rightarrow^{k+1} (M_k, w_k, s_k) \rightarrow (M', w', \perp)$. Then there exist M^*, w^*, s^* such that $(M, w_0, s) \rightarrow (M^*, w^*, s^*) \rightarrow^k (M_k, w_k, s_k) \rightarrow (M', w', \perp)$. Now, we have $\sup_k \Phi^k(\perp_\Lambda)(M, w_0, s) = \sup_k \Phi(\Phi^k(\perp_\Lambda))(M, w_0, s) = \sup_k \Phi^k(\perp_\Lambda)(M^*, w^*, s^*)$, and $\sup_k \Phi^k(\perp_\Lambda)(M^*, w^*, s^*) = M'$ by induction hypothesis, which ends the proof. ■

Corollary 15 *If $(M, 1, s) \Rightarrow (M_k, w_k, s_k) \rightarrow (M', w', \perp)$ and $s_k \neq \perp$, then $\text{peval}'(M, s) = M'$.*

Lemma 177 *If $\sup_n \Phi^n(\perp_\Lambda)(M, w_0, s) = M' \neq \text{fail}$, then either $s = \perp$ or $(M, w_0, s) \Rightarrow (M_k, w_k, s_k) \rightarrow (M', w', \perp)$ for some M_k, w_k, s_k, w' , where $s_k \neq \perp$.*

Proof: Like in lemma 164, for every M, w_0, s , we must have $\Phi^k(\perp_\Lambda)(M, w_0, s) = M'$ for some $k > 0$, and we can prove the result by induction on k .

- Base case. $k = 1$: we must have $s = \perp$ as otherwise we would have $M' = \text{fail}$.
- Induction step: suppose $\Phi^{k+1}(\perp_\Lambda)(M, w_0, s) = M'$. By definition of Φ , if $s \neq \perp$, we have $\Phi^k(\perp_\Lambda)(M^*, w^*, s^*) = M'$, where $g(M, w_0, s) \rightarrow (M^*, w^*, s^*)$. Since $M' \neq \text{fail}$ by assumption, Lemma 162 yields $(M, w_0, s) \rightarrow (M^*, w^*, s^*)$. By induction hypothesis, either $s^* = \perp$ or $(M^*, w^*, s^*) \Rightarrow (M^{**}, w^{**}, s^{**}) \rightarrow (M', w'', \perp)$ for some $M^{**}, w^{**}, s^{**}, w''$, where $s^{**} \neq \perp$. In the former case, we have $(M, w_0, s) \Rightarrow (M_k, w_k, s_k) \rightarrow (M', w', \perp)$ with $(M, w_0, s) = (M_k, w_k, s_k)$, $(M^*, w^*, s^*) = (M', w', \perp)$ and $s_k \neq \perp$, as required. In the latter case, we have $(M, w_0, s) \Rightarrow (M^{**}, w^{**}, s^{**}) \rightarrow (M', w'', \perp)$, with $s^{**} \neq \perp$. ■

Lemma 178 $\text{peval} = \text{peval}'$

Proof: We need to show that $\text{peval}(M, s) = \text{peval}'(M, s)$ for all $M \in C\Lambda$, $s \in \mathbb{U}$.

If $s = []$, then the equality follows trivially from the two definitions. Now, assume $s \neq []$.

If $\text{peval}'(M, s) = M' \neq \text{fail}$, then it follows from Lemma 177 that $\text{peval}(M, s) = M'$,

Now, let $\text{peval}'(M, s) = \text{fail}$ and suppose that $\text{peval}(M, s) = M' \neq \text{fail}$. Since $s \neq []$, by definition of peval there must be M_k, w_k, s_k, w' such that $(M, 1, s) \Rightarrow (M_k, w_k, s_k) \rightarrow (M', w', [])$ and $s_k \neq []$. But by Corollary 15, this implies that $\text{peval}'(M, s) = M' \neq \text{fail}$, which yields a contradiction. Hence $\text{peval}(M, s) = \text{fail}$. ■

Lemma 179 *For every k , $\text{peval}_k = \Phi^k(\perp^\lambda)$ is measurable.*

Proof: By induction on k :

- Base case: $k = 0$: $\text{peval}_0 = \perp^\lambda$ is a constant function on $C\Lambda \times \mathbb{U}$, so trivially measurable.
- Induction step : we have $\text{peval}_{k+1} = \Phi(\text{peval}_k)$, so it is enough to show that $\Phi(f)$ is measurable if f is measurable. $\Phi(f)$ is defined in pieces, so we want to use Lemma 139.

The domain of the first case is $C\Lambda \times \{[]\}$, so obviously measurable. The domain of the second case is $p^{-1}(g^{-1}(C\Lambda \times \mathbb{R} \times \mathbb{U}) \cap (C\Lambda \times \{1\} \times (\mathbb{U} \setminus \{[]\})))$, and $p(M, s) = (M, 1, s)$ is continuous, and so measurable. Hence, the domain is measurable. Finally, the domain of the last case is the complement of the union of the two above measurable sets, which means it is also measurable.

Thus, we only need to show that the functions corresponding to these three cases are measurable. This is obvious in the first and third case, because the corresponding functions are constant. The function for the second case is $\phi(M, s) = f(g(p(M, s)))$, where p is as defined above and g' is the restriction of g to $g^{-1}(C\Lambda \times \mathbb{R} \times \mathbb{U})$, which is measurable since restrictions preserve measurability. Since composition of measurable functions is measurable, ϕ is measurable.

Thus, peval_{k+1} is measurable, as required. ■

Lemma 180 *The function peval' is measurable $\mathcal{M}|_{C\Lambda} \times \mathcal{S} / \mathcal{M}|_{C\Lambda}$*

Proof: Corollary of Lemmas 179 and 166. ■

Restatement of Lemma 49 *The function peval is measurable $\mathcal{M}|_{C\Lambda} \times \mathcal{S} / \mathcal{M}|_{C\Lambda}$*

Proof: Corollary of Lemma 180 and Lemma 178. ■

E.5 Compositionality of peval

Lemma 181 *If $(M, 1, s) \Rightarrow (M_k, w_k, s_k) \rightarrow (M', w', [])$ for $s_k \neq []$ and $(M', 1, t) \Rightarrow (M_l, w_l, t_l) \rightarrow (M'', w''w'', [])$ for $s_k \neq []$, then $(M, 1, s@t) \Rightarrow (M_l, w'w_l, t_l) \rightarrow (M'', w'w'', [])$.*

Proof: By Lemma 37, we have $(M, 1, s@t) \Rightarrow (M_k, w_k, s_k@t) \rightarrow (M', w', t)$ and by Lemma 36, $(M', w', t) \Rightarrow (M_l, w'w_l, t_l) \rightarrow (M'', w'w'', [])$, so $(M, 1, s@t) \Rightarrow (M_l, w'w_l, t_l) \rightarrow (M'', w'w'', [])$. ■

Lemma 182 *If $(M, 1, s@t) \Rightarrow (M'', w'', [])$ and $M'' \neq \text{fail}$, then either $s = []$ or there exist unique $M_k, w_k, s_k \neq [], M' \neq \text{fail}, w', w'''$ such that $(M, 1, s) \Rightarrow (M_k, w_k, s_k) \rightarrow (M', w', [])$ and $(M', 1, t) \Rightarrow (M'', w''', [])$.*

Proof: By induction on the length of derivation of $(M, 1, s@t) \Rightarrow (M'', w'', [])$:

- Base case: $(M, 1, s@t) = (M'', w'', [])$. We have $s = []$, as required.
- Induction step: The result is trivial if $s = []$. Now, let us assume $s \neq []$. We have $(M, 1, s@t) \rightarrow (\hat{M}, \hat{w}, \hat{s}@t) \rightarrow^k (M'', w'', [])$ for some $\hat{M}, \hat{w}, \hat{s}$ and $k \geq 0$.

If $(M, 1, s@t) \rightarrow (\hat{M}, \hat{w}, \hat{s}@t)$ was derived with (RED RANDOM FAIL), then $\hat{M} = E[\text{fail}]$ for some E , which can only reduce to fail . Hence because $M'' \neq \text{fail}$ by assumption, we must have $k = 0$. We get $(M, 1, s) \rightarrow (E[\text{fail}], 0, [])$ and $(E[\text{fail}], 0, []) \rightarrow^0 (E[\text{fail}], 0, [])$, as required.

Now, assume $(M, 1, s@t) \rightarrow (\hat{M}, \hat{w}, \hat{s}@t)$ was not derived by (RED RANDOM FAIL). By Lemma 28, $\hat{w} > 0$.

- If $\hat{s} = []$, then by Lemma 59 we have $(M, 1, s) \rightarrow (\hat{M}, \hat{w}, [])$, so $(M, 1, s) \Rightarrow (M_k, w_k, s_k) \rightarrow (\hat{M}, \hat{w}, [])$ for $(M_k, w_k, s_k) = (M, 1, s)$. By Lemma 36, we have $(\hat{M}, 1, t) \rightarrow^k (M'', w''/\hat{w}, [])$, and $\hat{M} \neq \text{fail}$ follows from the fact that fail cannot reduce to $M'' \neq \text{fail}$. .
- If $\hat{s} \neq []$, then by Lemma 36, $(\hat{M}, 1, \hat{s}@t) \rightarrow^k (M'', w''/\hat{w}, [])$, so by the induction hypothesis, there exist $M_k, w_k, s_k \neq [], M' \neq \text{fail}, w', w'''$ such that $(\hat{M}, 1, \hat{s}) \Rightarrow (M_k, w_k, s_k) \rightarrow (M', w', [])$ and $(M', 1, t) \Rightarrow (M'', w''', [])$.
By Lemma 59, $(M, 1, s) \rightarrow (\hat{M}, \hat{w}, \hat{s})$, so by Lemma 36 we have $(M, 1, s) \rightarrow (\hat{M}, \hat{w}, \hat{s}) \Rightarrow (M_k, w_k \hat{w}, s_k) \rightarrow (M', w' \hat{w}, [])$ as required.

The uniqueness follows by Lemma 175 in Appendix E. ■

Restatement of Lemma 50 For all closed M, s, t , $\text{peval}(\text{peval}(M, s), t) = \text{peval}(M, s@t)$

Proof: If $s = []$ or $t = []$, the result follows immediately, because $\text{peval}(\text{peval}(M, []), t) = \text{peval}(M), t)$ and $\text{peval}(\text{peval}(M, s), []) = \text{peval}(M), s)$. Now, let us assume that $s \neq []$ and $t \neq []$.

If $(M, 1, s) \Rightarrow (M_k, w_k, s_k) \rightarrow (M', w', [])$ and $(M', 1, t) \Rightarrow (M_l, w_l, t_l) \rightarrow (M'', w'', [])$ for some $s_k, t_l \neq []$, then we have $\text{peval}(M, s) = M'$ and $\text{peval}(M', t) = M''$, so $\text{peval}(\text{peval}(M, s), t) = M''$. By Lemma 181, $(M, 1, s@t) \Rightarrow (M_l, w'w_l, t_l) \rightarrow (M'', w'w'', [])$, and so $\text{peval}(M, s@t) = M''$, as required.

If there are no M_k, w_k, s_k, M', w' such that $(M, 1, s) \Rightarrow (M_k, w_k, s_k) \rightarrow (M', w', [])$, then $\text{peval}(M, s) = \text{fail}$, and so $\text{peval}(\text{peval}(M, s), t) = \text{fail}$. Suppose for contradiction that $\text{peval}(M, s@t) = M'' \neq \text{fail}$. Then we must have $(M, 1, s@t) \Rightarrow (M_l, w_l, t_l) \rightarrow (M'', w'', [])$, for some M_l, w_l, w'' . But by Lemma 182, we have $(M, 1, s) \Rightarrow (M_k, w_k, s_k) \rightarrow (M', w', [])$ for some M_k, w_k, s_k, M', w' , which contradicts the assumption.

If we have $(M, 1, s) \Rightarrow (M_k, w_k, s_k) \rightarrow (M', w', [])$ but not $(M', 1, t) \Rightarrow (M_l, w_l, t_l) \rightarrow (M'', w'', [])$, then $\text{peval}(M, s) = M'$ and $\text{peval}(\text{peval}(M, s), t) = \text{peval}(M', t) = \text{fail}$. Again, suppose for contradiction that $\text{peval}(M, s@t) = M'' \neq \text{fail}$. We must have $(M, 1, s@t) \Rightarrow (M_l, w_l, t_l) \rightarrow (M'', w'', [])$, for some M_l, w_l, w'' . By Lemma 182 and Lemma 175, we have $(M, 1, s) \Rightarrow (M_k, w_k, s_k) \rightarrow (M', w', [])$ and $(M', 1, t) \Rightarrow (M'', w''', [])$ for some $M_k, w_k, s_k, M', w', w'''$. But by Lemma 182 (taking the second trace to be $[]$),

we have $(M', 1, t) \Rightarrow (M_l, w_l, t_l) \rightarrow (\hat{M}, \hat{w}, \square)$ for some $M_l, w_l, t_l, \hat{M} \neq \text{fail}, \hat{w}$, which contradicts the assumption. ■

E.6 Measurability of q and Q

Lemma 183 *If $M \Downarrow_w^\square G$ and $M \Downarrow_{w'}^s G'$, then $s = \square$.*

Proof: By induction on the derivation of $M \Downarrow_w^\square G$. ■

Lemma 184 *If $\mathbf{P}_M^{\mathcal{V}}(\square) > 0$, then $\mathbf{P}_M^{\mathcal{V}}(t) = 0$ for all $t \neq \square$.*

Proof: Follows directly from Lemma 183. ■

Lemma 185 (Tonelli's theorem for sums and integrals, 1.4.46 in [Tao, 2011]) *If (Ω, Σ, μ) is a measure space and f_1, f_2, \dots a sequence of non-negative measurable functions, then*

$$\int_{\Omega} \sum_{i=1}^{\infty} f_i(x) \mu(dx) = \sum_{i=1}^{\infty} \int_{\Omega} f_i(x) \mu(dx)$$

Proof: Follows from the monotone convergence theorem. ■

Lemma 186 (Linearity of Lebesgue integral, 1.4.37 ii) from [Tao, 2011]) *If (Ω, Σ) is a measurable space, f a non-negative measurable function, and μ_1, μ_2, \dots a sequence of measures on Σ , then*

$$\int_{\Omega} f(x) \sum_{i=1}^{\infty} \mu_i(dx) = \sum_{i=1}^{\infty} \int_{\Omega} f(x) \mu_i(dx)$$

Lemma 187 (Ex. 1.4.36 xi) from [Tao, 2011]) *If (Ω, Σ, μ) is a measure space and f a nonnegative measurable function on Ω and $B \in \Sigma$ and f^B a restriction of f to B , then*

$$\int_{\Omega} f(x) [x \in B] \mu(dx) = \int_B f(x) \mu^B(dx)$$

Below we write $q(s, t)$ as $q_M(s, t)$, to make the dependency on M explicit.

Let q_M^* be defined as follows:

$$q_M^*(s, t) = \begin{cases} \mathbf{P}_M^*([]) & \text{if } t = [] \\ q_M(s, t) & \text{otherwise} \end{cases}$$

Lemma 188 For all $M \in C\Lambda$ and $s, y \in \mathbb{U}$

$$q_M^*(s, t) = \begin{cases} \mathbf{P}_M^*(t) & \text{if } s = [] \text{ or } t = [] \\ \text{pdf}_{\text{Gaussian}}(s_1, \sigma^2, t_1) q_{\text{peval}(M, [s_1])}^*([s_2, \dots, s_{|s|}], [t_2, \dots, t_{|t|}]) & \text{otherwise} \end{cases}$$

Proof: By induction on $|s|$:

- Case $s = []$:

If $t = []$, the result follows directly from the definition of q_M^* . Otherwise, $q_M^*([], t) = q_M([], t) = P_M^*(t)$, as required.

- Case $|s| = n + 1 > 0$:

Again, if $t = []$, the result follows immediately. Otherwise, we have

$$q_M^*(s, t) = q_M(s, t) = \Pi_{i=1}^k (\text{pdf}_{\text{Gaussian}}(s_i, \sigma^2, t_i)) \mathbf{P}_{\text{peval}(M, [t_1, \dots, t_k])}^*(t)$$

where $k = \min(|s|, |t|) > 0$. Hence

$$\begin{aligned} q_M^*(s, t) &= \text{pdf}_{\text{Gaussian}}(s_1, \sigma^2, t_1) \Pi_{i=2}^k (\text{pdf}_{\text{Gaussian}}(s_i, \sigma^2, t_i)) \\ &\quad \mathbf{P}_{\text{peval}(M, [t_1, \dots, t_k])}^*([t_{k+1}, \dots, t_{|t|}]) \\ (\text{by Lemma 50}) &= \text{pdf}_{\text{Gaussian}}(s_1, \sigma^2, t_1) \Pi_{i=2}^k (\text{pdf}_{\text{Gaussian}}(s_i, \sigma^2, t_i)) \\ &\quad \mathbf{P}_{\text{peval}(\text{peval}(M, [t_1]), [t_2, \dots, t_k])}^*([t_{k+1}, \dots, t_{|t|}]) \\ &= (\text{pdf}_{\text{Gaussian}}(s_1, \sigma^2, t_1)) q_{\text{peval}(M, [s_1])}^*([s_2, \dots, s_{|s|}], [t_2, \dots, t_{|t|}]) \end{aligned}$$

as required. ■

Lemma 189 If $\mathbf{P}_M^*([]) > 0$, then $\text{peval}(M, t) = \text{fail}$ for every $t \neq []$.

Proof: It $\mathbf{P}_M^*([]) = w > 0$, then we must have $M \downarrow_w^\square V$ for some $V \in \mathcal{V}$, which implies $(M, 1, []) \rightsquigarrow^* (G, w, [])$. Using a lemma analogous to Lemma 30 for the score-ignoring reduction relation, we can easily show by induction that $(M, 1, t) \rightsquigarrow^* (G, w, t)$ for any $t \neq []$. Because the reduction relation is deterministic, this implies that there are no M', w' such that $(M, 1, t) \rightsquigarrow^* (M', w', [])$ (if there were, we would have $(M', w', []) \rightsquigarrow^* (G, w, t)$, but no reduction rule can add an element to a trace). This means that peval , by applying reduction repeatedly, will never reach $(M', [])$ for any M' , so $\text{peval}(M, t) = \text{fail}$. ■

Lemma 190 *If $\mathbf{P}_M^*([]) > 0$, then $q_M^*(s, t) = 0$ for all $s \in \mathbb{U}, t \neq []$.*

Proof: Follows easily from Lemma 189. ■

In order to prove measurability of the proposal density q , it is convenient to provide an alternative, fixpoint-based definition of \mathbf{P}_M^* , by means of a function \mathbf{P}'^* , analogous to \mathbf{P}' for \mathbf{P}_M .

Define:

$$\begin{aligned} h_{scr} & : \mathcal{T}_{scr} \rightarrow \mathcal{T} \\ h_{scr}(E[\text{score}(c)], w, s) & \triangleq (E[\text{true}], w, s) \end{aligned}$$

Lemma 191 *h_{scr} is measurable.*

Proof: Similar to the proof of measurability of g_{scr} . ■

$$\begin{aligned} h & : \mathcal{T} \rightarrow \mathcal{T} \\ h & \triangleq g_{val} \cup g_{det} \cup h_{scr} \cup g_{blocked} \end{aligned}$$

Lemma 192 *h is measurable.*

Proof: Identical to the proof of measurability of g , except that h_{scr} replaces g_{scr} . ■

$$\Psi_w(f)(M, w, s) \triangleq \begin{cases} w & \text{if } M \in \mathcal{V}, s = [] \\ f(h(M, w, s)) & \text{otherwise} \end{cases}$$

$$\mathbf{P}'^*(M, s) \triangleq \sup_n \Psi_w^n(\perp_w)(M, 1, s)$$

Lemma 193 \mathbf{P}'^* is measurable

Proof: Similar to the proof of measurability of \mathbf{P}' . ■

Lemma 194 $(M, w, s) \Rightarrow (G, w', [])$ for some w' if and only if $(M, w, s) \rightsquigarrow^* (G, w'', [])$ for some w'' .

Proof: The proof is a straightforward induction on the derivation of $(M, w, s) \Rightarrow (G, w', [])$. Details omitted. ■

Lemma 195 If $(M, w_0, s) \rightsquigarrow^* (V, w, [])$, then

$\sup_n \Psi_w^n(\perp_w)(M, w_0, s) = w$ and $\sup_n \Theta_\Lambda^n(\perp_\Lambda)(M, w_0, s) = V$.

Proof: Similar to the proof of Lemma 163. ■

Lemma 196 If $\sup_n \Psi_w^n(\perp_w)(M, w_0, s) = w \neq 0$, then $(M, w_0, s) \rightsquigarrow^* (V, w, [])$ for some $V \in \mathcal{V}$.

Proof: Similar to the proof of Lemma 164. ■

Lemma 197 For every $M \in C\Lambda$ and s , $\mathbf{P}'^*(M, s) = \mathbf{P}_M^*(s)$

Proof: If $(M, 1, s) \rightsquigarrow^* (V, w, [])$, then $\mathbf{P}_M^*(s) = w$ and $\mathbf{P}'^*(M, s) = \sup_n \Psi_w^n(\perp_w)(M, 1, s) = w$ by Lemma 195.

Now assume there are no V, w such that $(M, 1, s) \rightsquigarrow^* (V, w, [])$. Obviously, $\mathbf{P}_M^*(s) = 0$. If $\mathbf{P}'^*(M, s) = \sup_n \Psi_w^n(\perp_w)(M, 1, s) = w \geq 0$, then by Lemma 196, $(M, w_0, s) \rightsquigarrow^* (V, w, [])$ for some V, w , which contradicts the assumption. Hence, $\mathbf{P}'^*(M, s) = 0$. ■

Corollary 16 $\mathbf{P}_M^*(s)$ is measurable.

Lemma 198 $\int \mathbf{P}_M^*(s) ds \leq 1$

Proof: Similar to the proof of Lemma 173. ■

Lemma 199 For all $s \in \mathbb{U}$ and $M \in C\Lambda$, $\int_{\mathbb{U} \setminus \emptyset} q_M(s, t) \mu(dt) \leq 1$.

Proof: By induction on $|s|$.

- Base case: $s = \emptyset$

$$\begin{aligned} & \int_{\mathbb{U} \setminus \{\emptyset\}} q_M(\emptyset, t) \mu(dt) \\ &= \int_{\mathbb{U} \setminus \{\emptyset\}} \mathbf{P}_M^*(t) \mu(dt) \\ &\leq \int \mathbf{P}_M^*(t) \mu(dt) \\ &\text{by Lemma 198} \leq 1 \end{aligned}$$

- Induction step: $s \neq \emptyset$

We have:

$$\begin{aligned} & \int_{\mathbb{U} \setminus \{\emptyset\}} q_M(s, t) \mu(dt) \\ &= \int_{\mathbb{U} \setminus \{\emptyset\}} q_M^*(s, t) \mu(dt) \\ &= \sum_{i=1}^{\infty} \int_{\mathbb{R}^i} q_M^*(s, t) \mu(dt) \\ &\quad \text{(by Thm 16.9 from [Billingsley, 1995])} \\ &= \sum_{i=1}^{\infty} \int_{\mathbb{R}^i} q_M^*(s, t) \lambda^i(dt) \\ &\quad \text{(by Lemma 187)} \\ &= \sum_{i=1}^{\infty} \int_{\mathbb{R}^i} \text{pdf}_{\text{Gaussian}}(s_1, \sigma^2, t_1) q_{\text{peval}(M, [t_1])}^*([s_2, \dots, s_{|s|}], [t_2, \dots, t_{|t|}]) \lambda^i(dt) \\ &= \sum_{i=1}^{\infty} \int_{\mathbb{R}} \text{pdf}_{\text{Gaussian}}(s_1, \sigma^2, t_1) \int_{\mathbb{R}^{i-1}} q_{\text{peval}(M, [t_1])}^*(s', t') \lambda^{i-1}(dt') \lambda(dt_1) \\ &\quad \text{(by Fubini's theorem)} \\ &= \int_{\mathbb{R}} \text{pdf}_{\text{Gaussian}}(s_1, \sigma^2, t_1) \sum_{i=0}^{\infty} \int_{\mathbb{R}^i} q_{\text{peval}(M, [t_1])}^*(s', t') \lambda^i(dt') \lambda(dt_1) \\ &\quad \text{(by Lemma 185)} \\ &= \int_{\mathbb{R}} \text{pdf}_{\text{Gaussian}}(s_1, \sigma^2, t_1) \left(\int_{\{\emptyset\}} \mathbf{P}_{\text{peval}(M, [t_1])}^*(t') \mu(dt') + \right. \\ &\quad \left. \int_{\mathbb{U} \setminus \{\emptyset\}} q_{\text{peval}(M, [t_1])}^*(s', t') \mu(dt') \right) \lambda(dt_1) \end{aligned}$$

Now, we need to show that for all N ,

$$\int_{\{\emptyset\}} \mathbf{P}_N^*(t') \mu(dt') + \int_{\mathbb{U} \setminus \{\emptyset\}} q_N^*(s', t') \mu(dt') \leq 1 \quad (\text{E.1})$$

First, note that $\int_{\{\emptyset\}} \mathbf{P}_N^*(t') \mu(dt') \leq \int_{\mathbb{U}} \mathbf{P}_N^*(t') \mu(dt') \leq 1$, by Lemma 198. We also have $\int_{\{\emptyset\}} \mathbf{P}_N^*(t') \mu(dt') = \mathbf{P}_N^*(\emptyset)$, so by Lemma 190, if $\mathbf{P}_N^*(\emptyset) > 0$, then

$$\int_{\{\emptyset\}} \mathbf{P}_N^*(t') \mu(dt') + \int_{\mathbb{U} \setminus \{\emptyset\}} q_N^*(s', t') \mu(dt') = \int_{\{\emptyset\}} \mathbf{P}_N^*(t') \mu(dt') \leq 1$$

On the other hand, if $\mathbf{P}_N^*(\emptyset) = 0$, then

$$\begin{aligned} & \int_{\{\emptyset\}} \mathbf{P}_N^*(t') \mu(dt') + \int_{\mathbb{U} \setminus \{\emptyset\}} q_N^*(s', t') \mu(dt') \\ &= \int_{\mathbb{U} \setminus \{\emptyset\}} q_N^*(s', t') \mu(dt') = \int_{\mathbb{U} \setminus \{\emptyset\}} q_N(s', t') \mu(dt') \leq 1 \end{aligned}$$

by induction hypothesis.

Hence:

$$\begin{aligned} & \int_{\mathbb{U} \setminus \{\emptyset\}} q_M(s, t) \mu(dt) \\ & \leq \int_{\mathbb{R}} \text{pdf}_{\text{Gaussian}}(s_1, \sigma^2, t_1) \lambda(dt_1) \\ & = 1 \end{aligned}$$

as required. ■

Restatement of Lemma 52 For every $s, t \in \mathbb{U}$ and $M \in C\Lambda$, $q_M(s, t) \geq 0$.

Proof: Corollary of Lemma 199. ■

Restatement of Lemma 53 For any closed program M , the function q is measurable $\mathcal{S} \times \mathcal{S} / \mathcal{R}|_{\mathbb{R}_+}$.

Proof: It is enough to show that $q(s, t)$ is measurable for every $|s| = n$ and $|t| = m$, then the result follows from Lemma 139.

Note that a function taking a sequence s and returning any subsequence of it is trivially continuous and measurable, so for any function of s and t to be measurable, it is enough to show that it is measurable as a function of some projections of s and t .

- If $m > 0$ and $n < m$, then we have $q(s, t) = \prod_{i=1}^n \text{pdf}_{\text{Gaussian}}(s_i, \sigma^2, t_i) \mathbf{P}_{\text{peval}(M, t_{1..n})}^*(t_{n+1..m}) = \prod_{i=1}^n \text{pdf}_{\text{Gaussian}}(s_i, \sigma^2, t_i) \mathbf{P}'^*(\text{peval}(M, t_{1..n}), t_{n+1..m})$

Each $\text{pdf}_{\text{Gaussian}}(s_i, \sigma^2, t_i)$ is measurable, as a composition of a function projecting (s_i, t_i) from (s, t) and the Gaussian pdf, so their pointwise product must be measurable.

Now, \mathbf{P}'^* is measurable, and the function mapping (s, t) to $(\text{peval}(M, t_{1..n}), t_{n+1..m})$ is a pair of two measurable functions, one of which is a composition of the measurable $\text{peval}(M, \cdot)$ and a projection of $t_{1..n}$, and the other just a projection of $t_{n+1..m}$. Hence, the function mapping (s, t) to $\mathbf{P}'^*(\text{peval}(M, t_{1..n}), t_{n+1..m})$ is a composition of measurable functions.

Thus, $q(s, t)$ is a pointwise product of measurable functions, so it is measurable.

- If $m > 0$ and $n \geq m$, then $q(s, t) = \prod_{i=1}^m \text{pdf}_{\text{Gaussian}}(s_i, \sigma^2, t_i) \mathbf{P}_{\text{peval}(M, t)}^*(\square) = \prod_{i=1}^m \text{pdf}_{\text{Gaussian}}(s_i, \sigma^2, t_i) \mathbf{P}'^*(\text{peval}(M, t), \square)$

Now, the function mapping (s, t) to $\prod_{i=1}^m \text{pdf}_{\text{Gaussian}}(s_i, \sigma^2, t_i)$ is measurable like in the previous case. The function mapping (s, t) to $(\text{peval}(M, t), \square)$ is a pairing of two measurable functions, one being a composition of the projection of t and $\text{peval}(M, \cdot)$, the other being a constant function returning \square . Hence, $\mathbf{P}'^*(\text{peval}(M, t), \square)$ is a composition of two measurable functions. Thus, $q(s, t)$ is measurable.

- If $m = 0$, then $q(s, \square) = 1 - \int_{\mathbb{U} \setminus \{\square\}} q(s, t) \mu(dt)$. Since we have already shown that $q(s, t)$ is measurable on $\mathbb{U} \times (\mathbb{U} \setminus \{\square\})$, $\int_{\mathbb{U} \setminus \{\square\}} q(s, t) \mu(dt)$ is measurable by Fubini's theorem, so $q(s, \square)$ is a difference of measurable functions, and hence it is measurable.

■

Restatement of Lemma 54 *The function Q is a probability kernel on $(\mathbb{U}, \mathcal{S})$.*

Proof: We need to verify the two properties of probability kernels:

- (1) For every $s \in \mathbb{U}$, $Q(s, \cdot)$ is a probability distribution on \mathbb{U} . Since for every $s \in \mathbb{U}$, $q(s, \cdot)$ is non-negative measurable \mathcal{S} (by [Billingsley, 1995, Theorem 18.1]), $Q(s, B) = \int_B q(s, y) \mu(dy)$ (as a function of B) is a well-defined measure for all $s \in \mathbb{U}$. Finally, $Q(s, \mathbb{U}) = Q(s, \square) + Q(s, \mathbb{U} \setminus \{\square\}) = 1$.

(2) For every $B \in \mathcal{S}$, $Q(\cdot, B)$ is a non-negative measurable function on \mathbb{U} : Since $(\mathbb{U}, \mathcal{S}, \mu)$ is a σ -finite measure space, $q(\cdot, \cdot)$ is non-negative and measurable $\mathcal{S} \times \mathcal{S}$ and $Q(s, B) = \int_B q(s, y) \mu(ds)$, this follows from [Billingsley, 1995, Theorem 18.3].

■

References

- C. Andrieu, A. Doucet, and R. Holenstein. Particle Markov chain Monte Carlo methods. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 72(3):269–342, 2010. ISSN 1467-9868. doi: 10.1111/j.1467-9868.2009.00736.x. URL <http://dx.doi.org/10.1111/j.1467-9868.2009.00736.x>.
- N. Arora. Global seismic monitoring: A Bayesian approach, 2011. Presentation at the Twenty-Fifth Conference on Artificial Intelligence (AAAI-11), San Francisco, CA, USA.
- R. J. Aumann. Borel structures for function spaces. *Illinois J. Math.*, 5(4):614–630, 12 1961. URL <http://projecteuclid.org/euclid.ijm/1255631584>.
- D. Barber. *Bayesian Reasoning and Machine Learning*. Cambridge University Press, 2012.
- H. P. Barendregt. Handbook of logic in computer science (vol. 2). chapter Lambda Calculi with Types, pages 117–309. Oxford University Press, Inc., New York, NY, USA, 1992. ISBN 0-19-853761-1. URL <http://dl.acm.org/citation.cfm?id=162552.162561>.
- S. Bhat, J. Borgström, A. D. Gordon, and C. V. Russo. Deriving probability density functions from probabilistic functional programs. In N. Peterman and S. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS’13)*, volume 7795 of *Lecture Notes in Computer Science*, pages 508–522. Springer, 2013.
- P. Billingsley. *Probability and Measure*. Wiley, 3rd edition, 1995.
- A. Bizjak and L. Birkedal. Step-indexed logical relations for probability. In A. M. Pitts, editor, *Proceedings of FoSSaCS 2015*, volume 9034 of *LNCS*, pages 279–294. Springer, 2015.

- T. Blanc. *Propriétés de sécurité dans le lambda-calcul*. PhD thesis, Ecole Polytechnique, 2008. URL <http://moscova.inria.fr/~tblanc/memoire.pdf>.
- K. A. Bonawitz. *Composable Probabilistic Inference with Blaise*. PhD thesis, MIT, 2008. Available as Technical Report MIT-CSAIL-TR-2008-044.
- J. Borgström, A. D. Gordon, M. Greenberg, J. Margetson, and J. V. Gael. Measure transformer semantics for Bayesian machine learning. *Logical Methods in Computer Science*, 9(3), 2013. Preliminary version at ESOP’11.
- J. Borgström, A. D. Gordon, L. Ouyang, C. Russo, A. Ścibior, and M. Szymczak. Fabular: Regression formulas as probabilistic programming. Technical Report MSR–TR–2015–83, Microsoft Research, 2015.
- J. Borgström, A. D. Gordon, L. Ouyang, C. Russo, A. Ścibior, and M. Szymczak. Fabular: Regression formulas as probabilistic programming. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’16, pages 271–283, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3549-2. doi: 10.1145/2837614.2837653. URL <http://doi.acm.org/10.1145/2837614.2837653>.
- J. Borgström, U. D. Lago, A. D. Gordon, and M. Szymczak. A lambda-calculus foundation for universal probabilistic programming. In J. Garrigue, G. Keller, and E. Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 33–46. ACM, 2016. ISBN 978-1-4503-4219-3. doi: 10.1145/2951913.2951942. URL <http://doi.acm.org/10.1145/2951913.2951942>.
- G. V. d. Broeck, I. Thon, M. v. Otterlo, and L. D. Raedt. DTProbLog: A decision-theoretic probabilistic Prolog. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, AAAI’10, pages 1217–1222. AAAI Press, 2010. URL <http://dl.acm.org/citation.cfm?id=2898607.2898801>.
- Y. Cai. Asymptotic correctness criteria for the lightweight implementation of probabilistic programming. Unpublished manuscript, 2016.
- L. Cardelli. Type systems. In A. B. Tucker, editor, *The Computer Science and Engineering Handbook*, pages 2208–2236. CRC Press, 1997. ISBN 0-8493-2909-4.

- P. Cousot and M. Monerau. Probabilistic abstract interpretation. In H. Seidel, editor, *22nd European Symposium on Programming (ESOP 2012)*, volume 7211 of *Lecture Notes in Computer Science*, pages 166–190, Heidelberg, 2012. Springer.
- R. Crubillé and U. Dal Lago. Metric reasoning about lambda-terms: The general case. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, pages 341–367, 2017. doi: 10.1007/978-3-662-54434-1_13. URL https://doi.org/10.1007/978-3-662-54434-1_13.
- U. Dal Lago and M. Zorzi. Probabilistic operational semantics for the lambda calculus. *RAIRO-Theor. Inf. Appl.*, 46(3):413–450, 2012. doi: 10.1051/ita/2012012. URL <http://dx.doi.org/10.1051/ita/2012012>.
- V. Danos and R. Harmer. Probabilistic game semantics. *ACM Transactions on Computational Logic*, 3(3):359–382, 2002.
- V. Dorie. *Package blme, Reference Manual, Version 1.0-4*, 2016. URL <https://cran.r-project.org/web/packages/blme/index.html>.
- D. Duvenaud and J. Lloyd. Introduction to Probabilistic Programming. Talk at Computational and Biological Learning Lab, University of Cambridge. Slides available online at <http://mlg.eng.cam.ac.uk/Lloyd/talks/prob-prog-intro.pdf>, 2013.
- T. Ehrhard, C. Tasson, and M. Pagani. Probabilistic coherence spaces are fully abstract for probabilistic PCF. In *Proceedings of POPL 2014*, pages 309–320. ACM, 2014.
- T. S. Ferguson. A Bayesian analysis of some nonparametric problems. *Ann. Statist.*, 1(2):209–230, 03 1973. doi: 10.1214/aos/1176342360. URL <http://dx.doi.org/10.1214/aos/1176342360>.
- T. Gallay. *Théorie de la mesure et de l'intégration*. 2009. URL <http://im2ag-webmath.e.ujf-grenoble.fr/enseignement2/IMG/pdf/integrationa.pdf>. Course notes.
- A. Gelman and J. Hill. *Data Analysis Using Regression and Multilevel/Hierarchical Models*. Cambridge University Press, 2007.

- A. Gelman, D. Lee, and J. Guo. Stan: A probabilistic programming language for Bayesian inference and optimization. *Journal of Educational and Behavioral Statistics*, 40(5):530–543, 2015. doi: 10.3102/1076998615606113. URL <http://jeb.sagepub.com/content/40/5/530.abstract>.
- S. Geman and D. Geman. Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-6(6):721–741, Nov 1984. ISSN 0162-8828. doi: 10.1109/TPAMI.1984.4767596.
- A. Georgoulas, J. Hillston, and G. Sanguinetti. ABC-Fun: A probabilistic programming language for biology. In *Computational Methods in Systems Biology - 11th International Conference, CMSB 2013, Klosterneuburg, Austria, September 22-24, 2013. Proceedings*, pages 150–163, 2013. doi: 10.1007/978-3-642-40708-6_12. URL https://doi.org/10.1007/978-3-642-40708-6_12.
- A. Georgoulas, J. Hillston, D. Milios, and G. Sanguinetti. Probabilistic Programming Process Algebra. In *Quantitative Evaluation of Systems - 11th International Conference, QEST 2014, Florence, Italy, September 8-10, 2014. Proceedings*, pages 249–264, 2014. doi: 10.1007/978-3-319-10696-0_21. URL https://doi.org/10.1007/978-3-319-10696-0_21.
- W. R. Gilks, A. Thomas, and D. J. Spiegelhalter. A language and program for complex Bayesian modelling. *The Statistician*, 43:169–178, 1994.
- M. Giry. A categorical approach to probability theory. In B. Banaschewski, editor, *Categorical Aspects of Topology and Analysis*, volume 915 of *Lecture Notes in Mathematics*, pages 68–85. Springer Berlin Heidelberg, 1982. ISBN 978-3-540-11211-2. doi: 10.1007/BFb0092872. URL <http://dx.doi.org/10.1007/BFb0092872>.
- N. Goodman, V. K. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum. Church: a language for generative models. In *Uncertainty in Artificial Intelligence (UAI’08)*, pages 220–229. AUA Press, 2008.
- N. D. Goodman. The principles and practice of probabilistic programming. In *Principles of Programming Languages (POPL’13)*, pages 399–402, 2013.
- A. D. Gordon. An agenda for probabilistic programming: Usable, portable, and ubiquitous. Available online at <https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/fun-agendaforprobabilisticprogramming.pdf>, 2013.

- A. D. Gordon, M. Aizatulin, J. Borgström, G. Claret, T. Graepel, A. Nori, S. Rajamani, and C. Russo. A model-learner pattern for Bayesian reasoning. In *Principles of Programming Languages (POPL'13)*, 2013.
- A. D. Gordon, T. Graepel, N. Rolland, C. V. Russo, J. Borgström, and J. Guiver. Tabular: a schema-driven probabilistic programming language. In *Principles of Programming Languages (POPL'14)*, 2014.
- A. D. Gordon, C. V. Russo, M. Szymczak, J. Borgström, N. Rolland, T. Graepel, and D. Tarlow. Probabilistic programs as spreadsheet queries. In J. Vitek, editor, *Programming Languages and Systems (ESOP 2015)*, volume 9032 of *Lecture Notes in Computer Science*, pages 1–25. Springer, 2015.
- M. I. Gorinova. Probabilistic Programming with SlicStan, 2017. URL <http://homepages.inf.ed.ac.uk/s1207807/files/slicstan.pdf>. Master by Research dissertation, University of Edinburgh.
- R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In H. Boehm, B. Lang, and D. M. Yellin, editors, *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, USA, January 17-21, 1994*, pages 123–137. ACM Press, 1994. ISBN 0-89791-636-0. doi: 10.1145/174675.176927. URL <http://doi.acm.org/10.1145/174675.176927>.
- W. K. Hastings. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, 57(1):97–109, 1970.
- R. Herbrich, T. Minka, and T. Graepel. TrueSkilltm: A Bayesian skill rating system. In *Advances in Neural Information Processing Systems (NIPS'06)*, 2006.
- C. Heunen, O. Kammar, S. Staton, and H. Yang. A convenient category for higher-order probability theory. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*, pages 1–12, 2017. doi: 10.1109/LICS.2017.8005137. URL <https://doi.org/10.1109/LICS.2017.8005137>.
- M. D. Hoffman and A. Gelman. The no-U-turn sampler: Adaptively setting path lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research*, in press, 2013.

- J. Hölzl and A. Heller. *Three Chapters of Measure Theory in Isabelle/HOL*, pages 135–151. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-22863-6. doi: 10.1007/978-3-642-22863-6_12. URL https://doi.org/10.1007/978-3-642-22863-6_12.
- D. Huang and G. Morrisett. An application of computable distributions to the semantics of probabilistic programming languages. In P. Thiemann, editor, *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9632 of *Lecture Notes in Computer Science*, pages 337–363. Springer, 2016. ISBN 978-3-662-49497-4. doi: 10.1007/978-3-662-49498-1_14. URL https://doi.org/10.1007/978-3-662-49498-1_14.
- J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37: 67–111, 1998.
- C. Hur, A. V. Nori, S. K. Rajamani, and S. Samuel. Slicing probabilistic programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 133–144, 2014. doi: 10.1145/2594291.2594303. URL <http://doi.acm.org/10.1145/2594291.2594303>.
- C. Hur, A. V. Nori, S. K. Rajamani, and S. Samuel. A provably correct sampler for probabilistic programs. In P. Harsha and G. Ramalingam, editors, *Proceedings of FSTTCS 2015*, volume 45 of *LIPIcs*, pages 475–488. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015.
- D. Hutchison. ModelWizard: Toward interactive model construction. *CoRR*, abs/1604.04639, 2016. URL <http://arxiv.org/abs/1604.04639>.
- C. Jones. *Probabilistic non-determinism*. PhD thesis, University of Edinburgh, Edinburgh, Scotland, UK, 1989. URL <http://www.lfcs.inf.ed.ac.uk/reports/90/ECS-LFCS-90-105/>.
- C. Jones and G. D. Plotkin. A probabilistic powerdomain of evaluations. In *Logic in Computer Science (LICS'89)*, pages 186–195. IEEE Computer Society, 1989.

- O. Kiselyov. Problems of the lightweight implementation of probabilistic programming, 2016. Poster at PPS’2016 workshop.
- O. Kiselyov and C. Shan. Monolingual probabilistic programming using generalized coroutines. In *Uncertainty in Artificial Intelligence (UAI’09)*, 2009.
- D. Kozen. Semantics of probabilistic programs. *Journal of Computer and System Sciences*, 22(3):328–350, 1981.
- T. A. Le, A. G. Baydin, and F. Wood. Inference compilation and universal probabilistic programming. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 54 of *Proceedings of Machine Learning Research*, pages 1338–1348, Fort Lauderdale, FL, USA, 2017. PMLR.
- J.-J. Lévy. *Réductions correctes et optimales dans le lambda-calcul*. PhD thesis, Université Paris 7, 1978. URL <http://moscova.inria.fr/~levy/pubs/78phd.pdf>.
- D. Lunn, D. Spiegelhalter, A. Thomas, and N. Best. The BUGS project: Evolution, critique and future directions. *Statistics in Medicine*, 28(25):3049–3067, 2009. ISSN 1097-0258. doi: 10.1002/sim.3680. URL <http://dx.doi.org/10.1002/sim.3680>.
- D. J. C. MacKay. *Information Theory, Inference, and Learning Algorithms*. CUP, 2003.
- V. Mansinghka, D. Selsam, and Y. Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *CoRR*, 2014. arXiv:1404.0099v1 [cs.AI].
- V. K. Mansinghka, T. D. Kulkarni, Y. N. Perov, and J. B. Tenenbaum. Approximate Bayesian image interpretation using generative probabilistic graphics programs. Available at <http://arxiv.org/abs/1307.0060>, 2013.
- V. K. Mansinghka, R. Tibbetts, J. Baxter, P. Shafto, and B. Eaves. BayesDB: A probabilistic programming system for querying the probable implications of data. *CoRR*, abs/1512.05006, 2015. URL <http://arxiv.org/abs/1512.05006>.

- A. McCallum, K. Schultz, and S. Singh. FACTORIE: probabilistic programming via imperatively defined factor graphs. In *Advances in Neural Information Processing Systems 22: 23rd Annual Conference on Neural Information Processing Systems 2009. Proceedings of a meeting held 7-10 December 2009, Vancouver, British Columbia, Canada.*, pages 1249–1257, 2009.
- N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *Journal of Chemical Physics*, 21:1087–1092, 1953.
- B. Milch, B. Marthi, S. J. Russell, D. Sontag, D. L. Ong, and A. Kolobov. BLOG: Probabilistic models with unknown objects. In *Probabilistic, Logical and Relational Learning — A Further Synthesis*, 2005.
- T. Minka and J. Winn. Gates: A graphical notation for mixture models. Technical report, December 2008. URL <https://www.microsoft.com/en-us/research/publication/gates-a-graphical-notation-for-mixture-models/>.
- T. P. Minka. Expectation Propagation for approximate Bayesian inference. In *Uncertainty in Artificial Intelligence (UAI'01)*, pages 362–369. Morgan Kaufmann, 2001.
- C. C. Monnahan, J. T. Thorson, and T. A. Branch. Faster estimation of Bayesian models in ecology using Hamiltonian Monte Carlo. *Methods in Ecology and Evolution*, 8(3):339–348, 2017. ISSN 2041-210X. doi: 10.1111/2041-210X.12681. URL <http://dx.doi.org/10.1111/2041-210X.12681>.
- K. P. Murphy. *Machine learning: a probabilistic perspective*. MIT Press, Cambridge, MA, 2012.
- A. Nori, C.-K. Hur, and S. Rajamani. Semantics sensitive sampling for probabilistic programs. Technical report, October 2013. URL <https://www.microsoft.com/en-us/research/publication/semantics-sensitive-sampling-for-probabilistic-programs/>.
- A. Nori, S. Ozair, S. Rajamani, and D. a. Vijaykeerthy. Efficient synthesis of probabilistic programs. In *Programming Language Design and Implementation (PLDI)*. ACM—Association for Computing Machinery, June 2015. URL <https://www.microsoft.com/en-us/research/publication/efficient-synthesis-of-probabilistic-programs/>.

- A. V. Nori, C. Hur, S. K. Rajamani, and S. Samuel. R2: an efficient MCMC sampler for probabilistic programs. In C. E. Brodley and P. Stone, editors, *Proceedings of AAAI 2014*, pages 2476–2482. AAAI Press, 2014.
- E. Nummelin. *General Irreducible Markov Chains and Non-Negative Operators*. Cambridge Tracts in Mathematics ; no. 83. Cambridge University Press, Cambridge, 1984. ISBN 9780511526237.
- B. Paige and F. D. Wood. A compilation target for probabilistic programming languages. In *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014*, pages 1935–1943, 2014. URL <http://jmlr.org/proceedings/papers/v32/paige14.html>.
- S. Park, F. Pfenning, and S. Thrun. A probabilistic language based upon sampling functions. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 171–182, 2005. doi: 10.1145/1040305.1040320. URL <http://doi.acm.org/10.1145/1040305.1040320>.
- A. Pfeffer. IBAL: A probabilistic rational programming language. In B. Nebel, editor, *International Joint Conference on Artificial Intelligence (IJCAI’01)*, pages 733–740. Morgan Kaufmann, 2001.
- A. Pfeffer. Figaro: An object-oriented probabilistic programming language. Technical report, Charles River Analytics, 2009.
- A. M. Pitts. Alpha-structural recursion and induction. *Journal of the ACM*, 53:459–506, 2006. doi: <http://doi.acm.org/10.1145/1147954.1147961>.
- A. M. Pitts. *Nominal Sets: Names and Symmetry in Computer Science*, volume 57 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2013. ISBN 9781107017788.
- R. Pollack. Dependently typed records in type theory. *Formal Aspects of Computing*, 13:386–402, 2002. URL <http://homepages.inf.ed.ac.uk/rpollack/export/recordsFAC.ps.gz>.
- D. Purves and V. Lyutsarev. *Filzbach User Guide*, 2012. Available at <http://research.microsoft.com/en-us/um/cambridge/groups/science/tools/filzbach/filzbach.htm>.

- N. Ramsey and A. Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, pages 154–165, 2002. doi: 10.1145/503272.503288. URL <http://doi.acm.org/10.1145/503272.503288>.
- D. Ritchie, A. Stuhlmüller, and N. D. Goodman. C3: lightweight incrementalized MCMC for probabilistic programs using continuations and callsite caching. In *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics, AISTATS 2016, Cadiz, Spain, May 9-11, 2016*, pages 28–37, 2016. URL <http://jmlr.org/proceedings/papers/v51/ritchie16.html>.
- G. O. Roberts and J. S. Rosenthal. Harris recurrence of metropolis-within-gibbs and trans-dimensional markov chains. *Ann. Appl. Probab.*, 16(4):2123–2139, 11 2006. doi: 10.1214/1050516060000000510. URL <http://dx.doi.org/10.1214/1050516060000000510>.
- G. O. Roberts, J. S. Rosenthal, et al. General state space Markov chains and MCMC algorithms. *Probability Surveys*, 1:20–71, 2004.
- N. Saheb-Djahromi. *Probabilistic LCF*, pages 442–451. Springer Berlin Heidelberg, Berlin, Heidelberg, 1978. ISBN 978-3-540-35757-5. doi: 10.1007/3-540-08921-7_92. URL http://dx.doi.org/10.1007/3-540-08921-7_92.
- J. Schumann, T. Pressburger, E. Denney, W. Buntine, and B. Fischer. AutoBayes program synthesis system users manual. Technical Report NASA/TM–2008–215366, NASA Ames Research Center, 2008.
- A. Ścibior, Z. Ghahramani, and A. D. Gordon. Practical probabilistic programming with monads. In B. Lippmeier, editor, *Proceedings of Haskell 2015*, pages 165–176. ACM, 2015.
- M. Sperber, R. K. Dybvig, M. Flatt, and A. van Straaten. *Revised [6] report on the algorithmic language Scheme*, volume 19. Cambridge University Press, 2010.
- Stan Development Team. *Stan Modeling Language: User’s Guide and Reference Manual, Version 2.2*, 2014. URL <http://mc-stan.org/>.

- S. Staton, H. Yang, F. D. Wood, C. Heunen, and O. Kammar. Semantics for probabilistic programming: higher-order functions, continuous distributions, and soft constraints. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*, pages 525–534, 2016. doi: 10.1145/2933575.2935313. URL <http://doi.acm.org/10.1145/2933575.2935313>.
- A. Stuhlmüller and N. D. Goodman. A dynamic programming algorithm for inference in recursive probabilistic programs. *CoRR*, abs/1206.3555, 2012.
- T. Tao. *An Introduction to Measure Theory*. AMS, 2011.
- L. Tierney. Markov chains for exploring posterior distributions. *The Annals of Statistics*, 22(4):1701–1728, 1994.
- L. Tierney. A note on Metropolis-Hastings kernels for general state spaces. *The Annals of Applied Probability*, 8(1):1–9, 02 1998.
- D. Tolpin, J. van de Meent, and F. Wood. Probabilistic programming in Anglican. In A. Bifet, M. May, B. Zadrozny, R. Gavalda, D. Pedreschi, F. Bonchi, J. S. Cardoso, and M. Spiliopoulou, editors, *Proceedings of ECML PKDD 2015, Part III*, volume 9286 of *LNCS*, pages 308–311. Springer, 2015.
- N. Toronto. *Useful Languages for Probabilistic Modeling and Inference*. PhD thesis, Brigham Young University, Provo, UT, 2014. URL <https://www.cs.umd.edu/~ntoronto/papers/toronto-2014diss.pdf>.
- N. Toronto, J. McCarthy, and D. Van Horn. Running probabilistic programs backwards. In J. Vitek, editor, *Programming Languages and Systems: 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings*, pages 53–79, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. ISBN 978-3-662-46669-8. doi: 10.1007/978-3-662-46669-8_3. URL https://doi.org/10.1007/978-3-662-46669-8_3.
- D. Wingate, A. Stuhlmüller, and N. Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In *Proceedings of the 14th Intl. Conf. on Artificial Intelligence and Statistics*, page 131, 2011.

- J. Winn and T. Minka. Probabilistic programming with Infer.NET. Machine Learning Summer School lecture notes, available at <http://research.microsoft.com/~minka/papers/mlss2009/>, 2009.
- G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, MA, USA, 1993. ISBN 0-262-23169-7.
- F. Wood, J. W. van de Meent, and V. Mansinghka. A new approach to probabilistic programming inference. In *Proceedings of the 17th International conference on Artificial Intelligence and Statistics*, volume 33 of *JMLR Workshop and Conference Proceedings*, 2014. arXiv:1403.0504v2 [cs.AI].
- L. Yang, Y.-T. Yeh, N. D. Goodman, and P. Hanrahan. Incrementalizing McMC in probabilistic programs through tracing and slicing. 2013.
- L. Yang, P. Hanrahan, and N. D. Goodman. Generating efficient McMC kernels from probabilistic programs. In *AISTATS*, volume 33 of *JMLR Proceedings*, pages 1068–1076. JMLR.org, 2014.