SYNTACTIC FOUNDATIONS FOR MACHINE LEARNING

A Thesis
Presented to
The Academic Faculty

by

Sooraj Bhat

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing

Georgia Institute of Technology
May 2013

SYNTACTIC FOUNDATIONS FOR MACHINE LEARNING

Approved by:

Prof. Alexander Gray, Advisor
College of Computing
Georgia Institute of Technology

Prof. Charles Isbell, Advisor
College of Computing
Georgia Institute of Technology

Dr. Ashish Agarwal
Courant Institute of Mathematics
New York University

Prof. Richard Vuduc
College of Computing
Georgia Institute of Technology

Prof. Chung-chieh Shan
School of Informatics and Computing
Indiana University

Date Approved: 2 April 2013

ACKNOWLEDGEMENTS

I have been exceedingly fortunate to have met and interacted with a number of wonderful people who have made this thesis possible. I pray I have returned at least a fraction of the kindness and wisdom they have given to me.

My committee has been an unbelievable source of ideas and encouragement, and I am deeply grateful for their guidance. Rich Vuduc's door has always been open to me; time and time again he has given me sagely advice regarding my research and career. Ken Shan has been infectiously enthusiastic about my work, arming me with new ways of understanding it. I hope to learn more about how he sees our field.

Ashish Agarwal has been tremendously generous with his time, painstakingly initiating me in the ways of programming languages and type theory over countless emails and conversations. I cannot express how fortunate I feel to have had this luxury.

I am deeply indebted to my advisors Alex Gray and Charles Isbell. My conversations with Alex have taught me important lessons about myself and my relationship with research. He has been supportive of the foundational approach I have taken in this thesis and has provided thought-provoking perspectives that have profoundly influenced its outcome. Charles has been an incredible role model, and I have tried to keep mental notes on how to imitate him as a teacher, a researcher, a leader, a manager, a thinker, and a mentor. It is hard to convey how he has impacted me, nor the extent of my gratitude for him doing so.

I would also like to thank the many fellow graduate students and post-docs who have contributed to my research identity. I cherish our many collaborations and conversations about machine learning, research, and beyond. Many thanks to Michael Holmes, Chip Mappus, Josh Jones, David Minnen, Raffay Hamid, Santi Ontañón, Arya Irani, Julie Kientz, Shwetak Patel, David Roberts, Mark Nelson, Peng Zang, Liam Mac Dermed, Chris Simpkins, Jon Scholz, Maya Cakmak, Luis Cobo, Kaushik Subramanian, Neil Dantam, Arkadas Ozakin, Parikshit Ram, Ravi Ganti, Nishant Mehta, Ryan Riegel, and Bill March.

Many others have had a hand in this thesis. I have had the privilege of collaborating with Andrew Gordon, Claudio Russo, and Johannes Borgström on a portion of this work. Prof. Christopher Heil of the mathematics department provided valuable input when I was sorting out the theoretical foundations. Prof. Zhong Shao of Yale University graciously let me visit his lab for a summer, where I learned more about Coq. Wei Guan taught me about her mixed-integer SVM work so that I could implement it as an example in my thesis.

I have made several good friends at various stages of my graduate career. Although many have already come and gone from Atlanta, I want each of them know that their warmth is fondly remembered and their friendship is deeply appreciated.

Finally, I would like to thank my family for their amazing love and support. I would not have survived this thesis without the lessons my parents have taught me about perseverance. My sister has been an irreplaceable ally and confidante.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Machine learning has risen in importance across science, engineering, and business in recent years. Domain experts have begun to understand how their data analysis problems can be solved in a principled and efficient manner using methods from machine learning, with its simultaneous focus on statistical and computational concerns. Moreover, the data in many of these application domains has exploded in availability and scale, further underscoring the need for algorithms which find patterns and trends quickly and correctly.

However, most people actually analyzing data today operate far from the expert level. Available statistical libraries and even textbooks contain only a finite sample of the possibilities afforded by the underlying mathematical *principles*. Ideally, practitioners should be able to do what machine learning experts can do—employ the fundamental principles to experiment with the practically infinite number of possible customized statistical models as well as alternative algorithms for solving them, including advanced techniques for handling massive datasets. This would lead to more accurate models, the ability in some cases to analyze data that was previously intractable, and, if the experimentation can be greatly accelerated, huge gains in human productivity.

Fixing this state of affairs involves mechanizing and automating these statistical and algorithmic principles. This task has received little attention because we lack a suitable *syntactic* representation that is capable of specifying machine learning problems and solutions, so there is no way to encode the principles in question, which are themselves a mapping between problem and solution. This work focuses on providing the foundational layer for enabling this vision, with the thesis that such a representation is possible. We demonstrate the thesis by defining a syntactic representation of machine learning that is expressive, promotes correctness, and enables the mechanization of a wide variety of useful solution principles.

CHAPTER I

INTRODUCTION

Machine learning has risen in importance across science, engineering, and business in recent years. Domain experts have begun to understand how their data analysis problems can be solved in a principled and efficient manner using methods from machine learning, with its simultaneous focus on statistical and computational concerns. Moreover, the data in many of these application domains has exploded in availability and scale, further underscoring the need for algorithms which find patterns and trends quickly and correctly.

However, most people actually analyzing data today operate far from the expert level. Available statistical libraries and even textbooks contain only a finite sample of the possibilities afforded by the underlying mathematical *principles*. Ideally, practitioners should be able to do what machine learning experts can do—employ the fundamental principles to experiment with the practically infinite number of possible customized statistical models as well as alternative algorithms for solving them, including advanced techniques for handling massive datasets. This would lead to more accurate models, the ability in some cases to analyze data that was previously intractable, and, if the experimentation can be greatly accelerated, huge gains in human productivity.

Fixing this state of affairs involves mechanizing and automating these statistical and algorithmic principles. This task has received little attention because we lack a suitable *syntactic* representation that is capable of specifying machine learning problems and solutions, so there is no way to encode the principles in question, which are themselves a mapping between problem and solution. For instance, most representations used in current approaches omit the idea of optimization entirely, even though optimization is a cornerstone of machine learning, used in methods such as maximum likelihood estimation and support vector machines. This thesis focuses on providing the foundational layer for enabling this vision and proposes that such a suitable syntactic representation is possible.

1.1 Toward declarative machine learning

The idea of automatically deriving the implementation of a function from just a declarative specification of its input-output behavior is a holy grail in computer science. It is a central focus of many declarative domain-specific languages, such as SQL, as well as languages for *constraint programming*.

In this vein, there is currently a great need for automating the application of machine learning methods to new domain problems, from declarative specifications. Where does this need come from? Well, the methods in question are either

- **specific algorithms**, such as the C4.5 decision tree learning algorithm or support vector machines for performing supervised learning, or
- **general techniques/principles**, such as the *expectation maximization algorithm* for solving maximum likelihood problems, which is in fact an algorithm template: there is a different concrete instantiation for each different probabilistic model.

The research community has successfully translated the first category into software, as evidenced by the numerous libraries and toolkits for performing supervised and unsupervised learning [26, 49, 14, 53]. These packages are fairly straightforward to use in a black-box manner, requiring relatively little machine learning expertise from the user.

However, it is the latter category that contains the advanced methods for deriving efficient and customized algorithms for specific domain problems, which becomes relevant when off-the-shelf algorithms either (i) require the insertion of more domain knowledge to produce better quality answers, or (ii) are not computationally efficient enough. Unfortunately, these higher-level principles are currently relegated to *manual* pencil-and-paper derivations and can only be done by those with the necessary expertise. These techniques are *syntactic* in nature—if we could mechanize and automate them in a formal syntactic manner, users of machine learning would reap many benefits:

- **Declarative specifications, reduced expertise requirements.** Most domain experts are not experts in machine learning. Furthermore, machine learning is not a

single monolithic field of study but is instead composed of many areas of mathematics (such as probability, optimization, and linear algebra) as well as computer science (such as data structure design, parallel algorithms, and high-performance computing). Therefore, there is a high "barrier to entry" for those who wish to apply machine learning insights to their domain problems. Automation would eliminate this steep learning curve, allowing users to focus on the declarative formulation of their problems.

- **Reduced developer burden.** Automation would also eliminate wasteful redundancy that occurs in many current solutions. For instance, when performing Bayesian inference via *Markov chain Monte Carlo* (MCMC) methods, users often write their model twice: once in the sampling code they use for generating synthetic data from their model, and then again in the log-likelihood code, which is used to score candidate steps in MCMC's random walk. Keeping these two incarnations of the model in synch is tedious and error-prone. Automation would instead derive both codes from a single declarative specification, drastically reducing the amount of code the user needs to write, especially as the Kolmogorov complexity of the models increases. In particular, the user is saved from writing tricky log-likelihood code.

- **Verified algorithms.** One question that arises when manually deriving algorithms is: have I made a mistake somewhere? Fortunately, automation puts us on the path to mechanical verification of algorithm derivations. This is because a prerequisite to automating these techniques is a mechanization of the mathematics involved in expressing machine learning problems and solutions. Specifically, this requires formally treating probability distributions and optimization problems as syntactic objects. Because we are now treating machine learning problem as *programs*, we can leverage the large body of existing work on program verification from the programming language research community.

1.2 Limitations of current practice

The current mainstream practice for users interested in using machine learning techniques is to use existing libraries of statistical routines or collections of such libraries, such as the `R`

language and platform [53]. Users are limited to the specific models and algorithms provided by the library, and must write new code themselves to do learning on custom models, perhaps using library calls as subroutines where appropriate. The stochastic model and the learning task is implicit in the code written by the user; because there is no representation of these to pass to the library, the library cannot provide further assistance.

1.2.1 Probabilistic programming languages

The main issue with mainstream approaches is that they do not provide a *representation* of probabilistic models. If such a representation existed, the user could reify their models as data, which could then be passed to the library. This gives the libraries more information to work with. This is the approach taken by libraries such as the Bayes Net Toolbox [43] or Infer.NET [42], which use *graphical models* such as Bayesian networks or factor graphs as the data structure for representing stochastic models [33]. Here, the user builds a graph-based data structure encoding their model and then invokes the library to perform learning. Graphical models are a powerful unifying framework, capable of expressing a wide variety of models and amenable to generic inference algorithms. These approaches improve upon ordinary libraries by giving the user more flexibility in expressing models and saving them from the tedium of writing inference algorithms for each new custom model. However, it is up to the user to understand graphical models and to construct them properly. Furthermore, compared to the declarative notation of random variables, the interface is low-level, requiring users to explicitly construct graphs.

In response, *probabilistic programming languages* promise to arm data scientists with *declarative* languages for specifying their probabilistic models, while leaving the details of how to translate those models to efficient sampling or inference algorithms to a compiler. Ideally, a user would write their domain model as a short *probabilistic program*, such as

$$X \sim N(\mu, \sigma^2)$$
$$Y = e^X$$

and the system would be responsible for producing code for doing different things with that model. This model, for instance, introduces a normally distributed random variable X and

a random variable Y that is a deterministic transformation of X. A hypothetical compiler for such a language might produce the following `R` code when given this model:

```
# sampling code
x <- rnorm(100, mu, sigma);
y <- exp(x);


# probability density / likelihood
function(y, mu, sigma) {
  dnorm(log(y), mu, sigma) / y
}


# maximum likelihood estimation
function(y) {
  sum(log(y)) / length(y)
}
```

The first fragment samples synthetic data from our model. There is not much work for the compiler here, because the sampling code is in nearly one-to-one correspondence with the declarative specification. The next fragment is the parameterized *probability density function* of Y, computed in terms of the density of the normal distribution. This is an important concept in probability theory, and gives us the *likelihood function*, which is at the core of many machine learning methods. One such method is *maximum likelihood estimation*, and the last fragment is the closed-form solution for computing the maximum likelihood estimate of the parameter μ, given an array `y` of observations of the random variable Y. Notably, these latter two fragments require **syntactic operations** by the compiler. In particular, computing a closed-form solution for maximum likelihood estimates requires a closed-form (and therefore, syntactic) solution for the probability density function.

1.2.2 A missing collection of features

Unfortunately, this vision for probabilistic programming has not been fully realized in several key ways. In particular, no current language combines *all* of the following features:

ⓐ The ability to express optimization problems.

ⓑ The ability to express probability density functions.

ⓒ The ability to express arbitrary transformations of random variables.

ⓓ A formal language definition.

In fact, although the community as a whole has touched on each of these features, most existing works do not feature more than two of them simultaneously:

	ⓐ	ⓑ	ⓒ	ⓓ
Programming languages community				
Kozen [34], ...			✓	✓
EDSLs: Hansei [32], $\lambda_{\bigcirc}$ [48], ...			✓	✓
Infer.NET Fun [11]		✓	✓	✓
Machine learning community				
Church [23]			✓	✓
Markov Logic [56], ...		✓		✓
BUGS [37], HBC [15], ...		✓		
Infer.NET [42]		✓	✓	
Spiritual predecessors				
AutoBayes [20]	✓	✓		
Tyles [1]	✓			✓
This thesis	✓	✓	✓	✓

However, this collection of features is necessary if we are to practice machine learning as practitioners do and solve the problems that they solve. This thesis argues that this collection features is possible. Note that this is not as simple as stitching together features

from other languages—any time a new feature is added to a language, one must consider the possible feature interactions between the new feature and the existing features. In general, combining N features may require considering and resolving $O(N^2)$ feature interactions.

We now discuss why each of these features is needed and highlight challenges they pose for language design, implementation, and reasoning.

Expressing optimization problems. The concept of *numerical optimization*[1] (also known as *mathematical programming*) is pervasive in machine learning. Numerous learning frameworks in machine learning boil down to finding "best" instances according to some loss function. For example, learning in support vector machines is framed as finding the hyperplane that maximizes the margin between classes, measured in Euclidean distance. This is solved as a quadratic program. In maximum likelihood estimation, we search for parameters to a model which maximize its likelihood function. For simple models, this can be solved in closed form by symbolically manipulating KKT conditions, and for complex models by using the expectation maximization algorithm.

The importance of optimization is clear, but the real question is: why do we need optimization represented *in the language*? It is conceivable to have a probabilistic language with no intrinsic notion of optimization but where the compiler, external to the program, imposes a specific notion of optimization, *e.g.* the compiler would automatically derive maximum likelihood estimators for models defined in the language.

There are two major reasons to have an explicit notion of optimization in the language. First, it gives the user explicit control over what exact optimization they wish to express. For instance, the user may want to modify the standard maximum likelihood formulation, perhaps by adding an extra regularization term to enforce sparsity in the parameters or by adding constraints between parameters to reflect domain knowledge. Second, just as probabilistic programming seeks to free users from manually reformulating probabilistic programs, we wish to free users from manually reformulating optimization problems. There is a vast literature of such powerful reformulations, and systems such as Tyles [1, 2] and

[1]Not to be confused with *program optimization*, a compiler activity for producing efficient code.

ROSE [36] demonstrate the benefits of automating them.

A potential challenge is whether *random variables* and *optimization variables* can coexist in the same language. Both paradigms have subtly different interpretations and uses of variables. Optimization variables (those introduced by the min or max operator) have an existentially-quantified flavor: optimization seeks to find instantiations of the optimization variables. By contrast, random variables in probabilistic programs do not even behave as variables normally do. They are described formally as real-valued measurable functions [67], but then are used as real numbers—a type mismatch. They are bound to distributions (*e.g.* $X \sim N(0,1)$), but we cannot substitute them with their distributions in expressions (*e.g.* $X + 5$). Furthermore, data dependencies in a random variable's definition can affect answers, whereas ordinary evaluation usually only depends on a variable's value, not its data dependencies. Consider for example two random variables $X, Y \sim U(0,1)$ that are uniformly distributed on $[0,1]$. One might think that $\mathbb{P}(X+Y=0)$, the probability of their sum being equal to zero, is 0. However, if Y is defined as $Y = 1 - X$, this probability is in fact 1.

Expressing probability density functions. The *probability density function* f of a continuous distribution $\mathbb{P}$ is a convenient and compact characterization of the distribution. For real distributions, the relationship

$$\mathbb{P}(A) = \int_A f(x)\ dx$$

holds for every interval A of the real line; probabilities from the distribution are given by integrating the density function. Densities are a fundamental concept in machine learning, and parameterized densities give us *likelihood functions*, another important concept. Techniques which use density functions include maximum likelihood estimation, maximum a posteriori estimation, L_2 estimation, density-based Markov chain Monte Carlo sampling, and importance sampling.

Probabilistic programs correspond to probability distributions, which are distinct from density functions. We would like the compiler to automatically calculate density functions

from these programs. A major implementation challenge, however, is that not every distribution possesses a density function. This issue, in conjunction with the following feature, poses such a challenge that it is the topic of Chapter 2.

Arbitrary transformations of random variables. A common pattern in probabilistic models is the introduction of a series of of random variables bound to distributions, followed by more random variables defined as deterministic functions of the first group:

$$\begin{aligned} X &\sim N(0,1) & X_1 &\sim \mathbb{P}_1 \\ Y &= e^X & X_2 &\sim \mathbb{P}_2 \\ & & Y &= X_1 + X_2 \end{aligned}$$

This pattern arises in many applications, particular in the natural sciences, were there are specific functional models of how quantities interact. Furthermore, this capability automatically comes with the *probability monad*, which is a mathematical structure commonly used to organize probabilistic functional languages [55]. As we will see in Chapter 2, the probability monad is a desirable foundation and informs the formal study of such languages. This pattern is also useful for minimality and elegance of a language, because it gives us the power to define useful derived distributions without needing to add new primitive distributions to the language. For instance, the marginal distribution of Y in our running example $X \sim N(0,1), Y = e^X$ is a way to define the *log-normal distribution*.

Despite these attractive advantages, the expressivity of this features comes at a cost: it greatly complicates the calculation of density functions from probabilistic programs. This is why many existing systems, such as BUGS [37] and HBC [15], severely limit the user's ability to transform random variables, even prohibiting conceptually simple operations such as the addition of random variables.

A formal language definition. In this work, we use the phrase *formal language* to refer to a language defined by an abstract syntax, a type system, and a formal semantics. A natural question is: why are these parts necessary?

A type system gives us an understanding of the static behavior of programs. This includes classifying the nature of the inputs and outputs of operations in the language. This mechanism is important for us; for instance, asking for the maximum likelihood estimate for a model that does not have a density functions is an ill-posed question. Furthermore, the kinds of properties we wish to check are hard or impossible to check at run time. We cannot simply "run unit tests" as one might do in a dynamically typed language because we are working in the non-computable world of classical mathematics.

A formal semantics is desirable because it aids human and machine understanding of programs. This is particularly relevant when combining random variables and optimization variables. The notation of random variables is a convenient but informal shorthand for expressing machine learning problems that leans heavily on convention and a person's ability to resolve ambiguities—mechanization requires a much higher degree of precision. In particular, we would like that, for any legal program in our language, we can precisely identify the mathematical meaning of that program. This would help to debug program transformations, to state semantic preservation theorems, and to mechanically verify derivations of machine learning algorithms.

These reasons are also why we are interested in the probability monad, which informs the design of the type system and semantics of a probabilistic language.

1.3 Thesis statement

In light of this discussion, we argue that

> It is possible to construct a syntactic representation of machine learning that is expressive, promotes correctness, and can mechanize useful solution principles.

There are three subclaims:

- **Expressive.** Being expressive means lowering the semantic gap between the mathematical statement of a learning problem and the code needed to express it. Small conceptual changes to a learning problem should result in only small code changes to

the corresponding program.

Validation: We validate this part of the thesis by first defining an expressive language for probability (Chapter 2) and then combining it with Tyles [1, 2], a language for optimization (reviewed in Chapter 3), to provide an expressive representation of machine learning, implemented as a theory in the Coq proof assistant (Chapter 4). Chapter 2 demonstrates expressivity by providing the first-ever syntactic theory of *probability density functions* that solves the issue of supporting an operator for expressing the density functions of probabilistic programs in the context of highly expressive probabilistic programming languages. Chapter 4 demonstrates expressivity by presenting programs that utilize optimization constructs in addition to probability constructs for expressing important machine learning problems, such as maximum likelihood estimation and sparse support vector machines.

- **Promotes correctness.** Promoting correctness includes reducing or eliminating the possibility of the user writing ill-formed programs, supporting the ability of a human or computer to reason about programs, and providing a pathway to mechanical verification of program transformations.

  Validation: We validate this part of the thesis by using *type theory* to formally define the probability language in Chapter 2 and to inform our design of the representation for machine learning in Chapter 4. Type theory is a modern metamathematical framework with deep ties to both mathematics and programming languages. It disciplines our language design, obligating us to precisely capture the semantics of constructs such as probability distributions, and it promotes correctness by ruling out ill-formed programs by definition. We demonstrate these qualities with the language definition in Chapter 2, where the type system for density functions is a notable contribution.

  Furthermore, we wish to support systems in which it is possible to mechanically prove program transformations correct, rather than trusting their authors. Types are the path to constructing proofs, as demonstrated by modern theorem proving systems that are grounded in advanced dependent type theories, such as Coq [6]. We show

in Chapter 4 how program transformations written using our formalization can be encoded as equality theorems and then used for program rewriting via existing Coq mechanisms. In this setup, algorithm derivation reduces to proof search. Such equality theorems can be initially stated as axioms for the purposes of rapid prototyping, while saving actual proof writing for a later stage, retaining the option of full mechanical verification.

- **Can mechanize useful solution principles.** This is achieved by taking solution principles for converting learning problems into algorithms and encoding them into a mechanical form. We focus on methods of a syntactic nature, which are more resistant to mechanization and are not easily implemented as ordinary libraries. A method is *useful* if it is widely used, well-established, or a recent research result.

  Validation: We validate this part of the thesis by encoding several solution principles and techniques as program transformation using the representations developed throughout the dissertation.

  - *Computing probability density functions.* We define a compiler for computing the probability density function of a probabilistic program. We show that the compiler is useful by applying it to inference problems from ecology. We use the compiler to compute the log-likelihood function of a probabilistic model, which is then used for Bayesian inference via Markov chain Monte Carlo sampling (Chapter 2).
  - *The big-M method for disjunctive constraints.* Agarwal defines a compiler that implements the convex-hull method for transforming disjunctive constraints in optimization problems [1]. We define another such compiler that employs the *big-M method* [54]. We apply both compilers to problems from chemical process engineering and operations research, showing that experimentation is accelerated and that performance is competitive with human experts and state-of-the-art optimization software (Chapter 3).
  - *The big-M method for L_0 regularization.* Likewise, we encode a transformation

for optimization problems that contain L_0 regularization terms, which are often added in machine learning problems to induce solution sparsity. We show how this transformation can be used on the L_0 *support vector machine* formulation to recreate *mixed-integer support vector machines* [25] (Chapter 4).

– *Expectation maximization.* Finally, we encode the expectation maximization algorithm [17] for solving maximum likelihood estimation problems. This technique makes fundamental use of both probability and optimization constructs in its formulation; our ability to handle this is a major feature of this work. We report on our Coq implementation (Chapter 4).

CHAPTER II

A SYNTACTIC THEORY OF PROBABILITY

There has been great interest in creating probabilistic programming languages to simplify the coding of statistical tasks; however, there still does not exist a formal language that simultaneously provides (i) continuous probability distributions, (ii) the ability to naturally express custom probabilistic models, and (iii) probability density functions (PDFs). This collection of features is necessary for mechanizing fundamental statistical techniques. We formalize the first probabilistic language that exhibits these features, and it serves as a foundational framework for extending the ideas to more general languages. Particularly novel are our type system for *absolutely continuous* (AC) distributions (those which permit PDFs) and our PDF calculation procedure, which calculates PDFs for a large class of AC distributions. Our formalization paves the way toward the rigorous encoding of powerful statistical reformulations.

2.1 Introduction

In the face of more complex data analysis needs, both the machine learning and programming languages communities have recognized the need to express probabilistic and statistical computations *declaratively*. This has led to a proliferation of probabilistic programming languages [48, 19, 23, 31, 32, 34, 41, 50, 55, 56, 57]. Program transformations on probabilistic programs are crucial: many techniques for converting statistical problems into efficient, executable algorithms are syntactic in nature. A rigorous language definition aids reasoning about the correctness of these program transformations.

However, several fundamental statistical techniques cannot currently be encoded as program transformations because current languages have weak support for probability distributions on continuous or hybrid discrete-continuous spaces. In particular, no existing language rigorously supports expressing the *probability density function* (PDF) of custom

probability distributions. This is an impediment to mechanizing statistics; continuous distributions and their PDFs are ubiquitous in statistical theory and applications. Techniques such as maximum likelihood estimation (MLE), L_2 estimation (L2E), and nonparametric kernel methods are all formulated in terms of the PDF [9, 59, 60]. Specifically, we want the ability to naturally express a probabilistic model over a discrete, continuous or hybrid space and then mechanically obtain a usable form of its PDF. Usage of the PDF may entail direct numerical evaluation of the PDF or symbolic manipulation of the PDF and its derivatives. Continuous spaces pose some unique obstacles, however. First, the existence of the PDF is not guaranteed, unlike the discrete case. Second, stating the conditions for existence involves the language of measure theory, an area of mathematics renowned for nonconstructive results, suggesting that mechanization may not be straightforward. Notably, obtaining a PDF from its distribution is a non-computable operation in the general case [28]. In light of these issues, we make the following new contributions:

- We present a formal probability language with classical measure-theoretic semantics which allows naturally expressing a variety of useful probability distributions on discrete, continuous and hybrid discrete-continuous spaces, as well as their PDFs when they exist (Section 2.3). The language is a core calculus which omits functions and mutation.

- We define a type system for *absolutely continuous* probability distributions, *i.e.* those which permit a PDF. The type system does not require mechanizing σ-algebras, null sets, the Lebesgue measure, or other complex constructions from measure theory. The key insight is to analyze a distribution by how it transforms other distributions instead of using the "obvious" induction on the monadic structure of the distribution (Section 2.4).

- We define a procedure that calculates PDFs for a large class of distributions accepted by our type system. The design permits modularly adding knowledge about individual distributions with known PDFs (but which cannot be calculated from scratch),

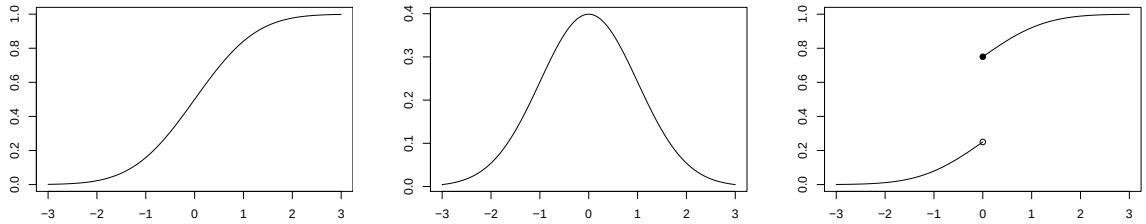


Figure 1: The CDF and PDF of a standard normal distribution and the CDF of a distribution that does not have a PDF.

enabling the procedure to proceed with programs that use these distributions as subcomponents (Section 2.5).

We believe this is the first general treatment of PDFs in a language. We deliberately omit features that are not essential to the current investigation (*e.g.* expectation, sampling). Finally, we discuss the relation to existing and future work (Sections 2.7 and 2.8). In particular, we save a treatment of PDFs in the context of conditional probability for future work.

2.2 Background and motivation

We first introduce probability in the context of countable spaces to emphasize the complications that arise when moving to continuous spaces. We focus only on issues surrounding PDFs. We occasionally deviate from standard probability notation to circumvent imprecision in the standard notation and to create a harmonious notation throughout the paper. In this section we present a specialized account of probability for ease of exposition. We discuss the rigorous and generalized definitions in Section 2.3.3.

We use the term *discrete distribution* for distributions on discrete spaces (countable sets); *continuous distribution* for distributions on the continuous spaces $\mathbb{R}$ and $\mathbb{R}^n$; and *hybrid distribution* for distributions on products of discrete and continuous spaces that are themselves neither discrete nor continuous, such as $\mathbb{R} \times \mathbb{Z}$.

2.2.1 Probability on countable spaces

Consider a set of outcomes A. For now, let A be countable. It is meant to represent the possible states of the world we are modeling, such as the set of possible outcomes of an

experiment or measurements of a quantity. An *event* is a subset of A, also understood as a predicate on elements of A. Events denote some occurrence of interest and partition the outcomes into those that exhibit the property and those that do not. A *probability distribution* $\mathbb{P}$ (or simply, *distribution*) on A is a function from events to $[0, 1]$ such that $\mathbb{P}(X) \geq 0$ for all events X, $\mathbb{P}(A) = 1$ and $\mathbb{P}(\bigcup_{i=0}^{\infty} X_i) = \sum_{i=0}^{\infty} \mathbb{P}(X_i)$ for countable sequences of mutually disjoint events X_i. Distributions tell us the probability that different events will occur. It is generally more convenient to work with a distribution's *probability mass function* (PMF) instead, defined $f(x) = \mathbb{P}(\{x\})$, which tells us how likely an individual outcome is. It satisfies

$$\mathbb{P}(X) = \sum_{x \in X} f(x)$$

for all events X on A. For example, if $\mathbb{P}$ is the distribution characterizing the outcome of rolling a fair die, its PMF is given by $f(x) = \frac{1}{6}$, where $x \in A$ and $A = \{1, 2, 3, 4, 5, 6\}$. The probability an even number is rolled is $\mathbb{P}(\{2, 4, 6\}) = \frac{1}{6} + \frac{1}{6} + \frac{1}{6} = \frac{1}{2}$.

2.2.2 Moving to continuous spaces

A *probability density function* (PDF) is the continuous analog of the PMF. Unfortunately, although every distribution on a countable set has a PMF, not every distribution on a continuous space has a PDF. Consider distributions on the real line. We say that a function f is a PDF of a distribution $\mathbb{P}$ on $\mathbb{R}$ if for all events X,

$$\mathbb{P}(X) = \int_X f(x)\ dx, \tag{1}$$

which states that the probability of X is the integral of f on X (in the simplest case, X is an interval). This idea can be extended to more general spaces. This equation does not determine f uniquely, but any two solutions f_1 and f_2 are equal "almost everywhere" (see Section 2.3.3) and give identical results under integration. Thus, we often refer to a PDF as *the* PDF.

For the spaces we consider in this paper, the property of having a PDF is equivalent to being *absolutely continuous* (AC). Roughly speaking, a distribution is AC if it never assigns positive probability to events that have "size zero" in the underlying space. For

instance, the standard normal distribution is absolutely continuous and has the PDF $\phi(x) = \exp(-x^2/2)/\sqrt{2\pi}$. On the other hand, the distribution of y in the model

$$\begin{array}{l} z \sim \text{CoinFlip}(1/2) \\ x \sim \text{Normal}(0,1) \end{array} \qquad y = \begin{cases} 0 & \text{if } z = \text{heads} \\ x & \text{if } z = \text{tails.} \end{cases}$$

does not have a PDF. We have used *random variables* to write the model; this is a commonly used informal notation that is shorthand for a more rigorous expression that defines the model. The model represents the following process: flip a fair coin; return 0 if it is heads, and sample from the standard normal distribution otherwise. We can see that it is not AC: the event $\{0\}$ occurs with probability $1/2$ (whenever the coin comes up heads), but has an interval length of zero. We use the *cumulative distribution function* (CDF) to visualize each distribution (Figure 1); the CDF F of a distribution $\mathbb{P}$ on $\mathbb{R}$ is $F(x) = \mathbb{P}(\ (-\infty, x]\ )$ and gives the probability that a sample from the distribution takes a value less than or equal to x. From Equation 1 we know that $\mathbb{P}$ has a PDF if and only if there exists a function f such that $F(x) = \int_{-\infty}^{x} f(t)\ dt$. Clearly, no such function exists for the CDF of y due to the jump discontinuity.

Mixing discrete and continuous types is not the only culprit. Consider the following process: sample a number u uniformly randomly from $[0, 1]$ and return the vector $x = (u, u) \in \mathbb{R}^2$. The distribution of x (a distribution on $\mathbb{R}^2$) is not AC: the event $X = \{(u, u) \mid u \in [0, 1]\}$ has probability 1, but X is a line segment and thus has zero area. Likewise, there is no PDF on $\mathbb{R}^2$ we could integrate to give positive mass on this zero area line segment.

2.2.3 Applications of the pdf

The PDF is often used to compute expectations (and probabilities, which are a special case of expectation). Expectation is a fundamental operation in probability and is used in defining quantities such as mean and variance. The expectation operation $\mathbb{E}$ of a distribution $\mathbb{P}$ on $\mathbb{R}$ is a higher-order function that satisfies

$$\mathbb{E}(g) = \int_{-\infty}^{\infty} g(x) \cdot f(x)\ dx$$

when $\mathbb{P}$ has a PDF, f, and the integral exists. Another application is *maximum likelihood estimation* (MLE), which addresses the problem of choosing the member of a family of distributions that "best explains" observed data. Let $\mathbb{P}(\cdot\,;\cdot)$ be a parameterized family of distributions, where $\mathbb{P}(\cdot\,;\theta)$ is the distribution for a given parameter θ. The MLE estimate θ^* of $\mathbb{P}$ for observed data x is given by

$$\theta^* = \arg\max_\theta f(x;\theta)$$

where $f(\cdot\,;\theta)$ is the PDF of $\mathbb{P}(\cdot\,;\theta)$. For example, x could be a set of points in $\mathbb{R}^n$ we wish to cluster, and θ^* could be the estimate of the locations of the cluster centroids. $\mathbb{P}$ would be the family of distributions we believe generated the clusters (a family parameterized by the positions of the cluster centroids), such as a mixture-of-Gaussians model. More details are available in [9].

2.2.4 Challenges for language design

Categorically, probability distributions form a monad [22, 55]. This structure forms the basis of many probabilistic languages because it is minimal, elegant, and presents many attractive features. First, it provides the look and feel of informal random variable notation, allowing us to express models as we normally would, while remaining mathematically rigorous. The monad structure affords formulating probability as an embedded domain specific language [19, 48, 55, 32] or as a mathematical theory in a proof assistant [3]. Additionally, many proofs about distributions expressed in the probability monad are greatly simplified by the monadic structure. We feel it is desirable to structure a language around the probability monad, and we investigate supporting PDFs specifically in such languages.

The probability monad consists of monadic return and monadic bind, as usual. Monadic return corresponds to the point mass distribution. We also provide the Uniform(0,1) distribution as a monadic value. These three combinators can be used to express a variety of distributions. The main issue when designing a type system for absolute continuity is that return creates non-AC distributions (on continuous types), yet—as a core member of the monadic calculus—it appears in the specification of nearly every distribution, even those that *are* AC. The "obvious" induction along the monadic structure is difficult to use to prove

absolute continuity in the cases of interest. Consider for instance the joint distribution of two independent Uniform(0,1) random variables, written in our language as

$$\mathsf{var}\ x \sim \mathsf{random}\ \mathsf{in}\ \mathsf{var}\ y \sim \mathsf{random}\ \mathsf{in}\ \mathsf{return}\ (x, y). \tag{2}$$

It is AC even though the subexpressions return (x, y) and var $y \sim$ random in return (x, y) are both *not* AC, where we treat x and y as real numbers, as dictated by the typing rule for bind. Also, as implementors we have found it difficult to "eyeball" the rules for absolute continuity. For example, only the first of these distributions is AC even though they are all nearly identical to Equation 2:

$$\mathsf{var}\ x \sim \mathsf{random}\ \mathsf{in}\ \mathsf{var}\ y \sim \mathsf{random}\ \mathsf{in}\ \mathsf{return}\ (x, x + y)$$

$$\mathsf{var}\ x \sim \mathsf{random}\ \mathsf{in}\ \mathsf{var}\ y \sim \mathsf{random}\ \mathsf{in}\ \mathsf{return}\ (x, y - y)$$

$$\mathsf{var}\ x \sim \mathsf{random}\ \mathsf{in}\ \mathsf{var}\ y \sim \mathsf{random}\ \mathsf{in}\ \mathsf{return}\ (x, y, x + y)$$

Clearly, what is needed is a principled analysis. We provide this in Section 2.4. A natural urge is wanting to remove return to create a language in which only AC distributions are expressible. We feel this is undesirable. Without return, we would not be able to express something as simple as adding two random variables (consider $(x + y)$ instead of (x, y) in Equation 2). Essentially, return allows us to express random variables as transformations of other random variables—a fundamental modeling tool we feel should be supported, allowing users to write down models that most naturally capture their domain. Without return we must extend the core calculus for each transformation we wish to use on random variables, and we must do so carefully if we want to ensure that non-AC distributions remain inexpressible. This extension of the core detracts from minimality and elegance, and it complicates developing the theory in a verification environment such as Coq, one of our eventual goals.

Finally, in addition to checking for existence, we would like to also calculate a usable form for the PDF. Many current probabilistic languages focus on distributions with only *finitely* many alternatives, which allows for implementing distributions as weighted lists of outcomes. The probability monad in this case is similar to the list monad, with some added

$$
\begin{array}{rl}
\text{Variables} & x, y, z, u, v \qquad \text{Literals} \quad l \\
\text{Base types} & \tau ::= \mathsf{bool} \mid \mathsf{Z} \mid \mathsf{R} \mid \tau_1 \times \tau_2 \\
\text{Types} & t ::= \tau \mid \mathsf{dist}\ \tau \\
\text{Expressions} & \varepsilon ::= x \mid l \mid op\ \ \varepsilon_1 ... \varepsilon_n \mid \mathsf{if}\ \varepsilon_1\ \mathsf{then}\ \varepsilon_2\ \mathsf{else}\ \varepsilon_3 \\
\text{Primops} & op ::= + \mid * \mid \mathsf{neg} \mid \mathsf{inv} \mid = \mid < \mid (\cdot, \cdot) \mid \mathsf{fst} \mid \mathsf{snd} \\
 & \quad \mid \mathsf{exp} \mid \mathsf{log} \mid \mathsf{sin} \mid \mathsf{cos} \mid \mathsf{tan} \mid \mathsf{R\_of\_Z} \\
\text{Distributions} & e ::= \mathsf{random} \mid \mathsf{return}\ \varepsilon \mid \mathsf{var}\ x \sim e_1\ \mathsf{in}\ e_2 \\
\text{Programs} & p ::= \mathsf{pdf}\ e \\
\text{Contexts} & \Gamma ::= \emptyset \mid \Gamma, x : \tau \qquad \Upsilon ::= \emptyset \mid \Upsilon, x : \tau \sim e \\
\text{Substitution} & e[x := \varepsilon] \qquad \text{Free variables} \quad FV(\,\cdot\,)
\end{array}
$$

Figure 2: The abstract syntax.

logic describing how bind should propagate the weights. The weighted lists correspond directly to the PMF, but no such straightforward computational strategy exists for the PDF. We explore this further in Section 2.5.

2.3 The language

In this section we present the abstract syntax, type system and semantics for our probabilistic language, except for the parts related to PDFs, which we cover in Section 2.4.

2.3.1 Abstract syntax

Figure 2 contains the syntax definitions. In addition to the standard letters for variables, we also use u and v when we want to emphasize that a random variable is distributed according to the Uniform(0,1) distribution. The syntactic category for literals includes Boolean (bool), integer (Z), and real number (R) literals. Types are stratified to ensure that distributions (dist τ) are only over base types. Integers are a distinct type from the reals; there is no subtyping in the language. We also stratify terms to simplify analysis. Expressions and primitive operations (primops) take their standard mathematical meaning, unless noted otherwise. For simplicity, we overload addition, multiplication, negation, and integer literals on the integers and reals, but fundamentally there is a + for integers and a separate + for reals, *etc.* Inversion inv denotes the reciprocal operation, and log is the

natural logarithm. We give our semantics in terms of classical mathematics, so we do not concern ourselves with the issue of computation on the reals. Equality is defined on all base types in the usual way, and less-than is defined only on the numeric types. We write $(\varepsilon_1, \varepsilon_2, ..., \varepsilon_{n-1}, \varepsilon_n)$ as shorthand for $(\varepsilon_1, (\varepsilon_2, ...(\varepsilon_{n-1}, \varepsilon_n)...))$. The function R_of_Z injects an integer into the reals. The distribution random corresponds to the Uniform(0,1) distribution. The next two constructs correspond to monadic return and bind for the probability monad. The distribution return ε is the *point mass distribution*, which assigns probability 1 to the event $\{\varepsilon\}$. A random variable distributed according to return ε is in fact deterministic: there is no variation in the value it can take. The bind construct, var $x \sim e_1$ in e_2, is used to build complex distributions from simpler ones. It can be read: "introduce random variable x, distributed according to the distribution e_1, with scope in the distribution e_2". It is the only binding construct in the language. For simplicity, we have chosen to omit let-bindings and functions from our language, but we use both in our examples. We can use standard substitution rules to reduce such examples to the syntax of Figure 2. Examples include $\langle\varepsilon\rangle :=$ if ε then 1 else 0 (to inject Booleans into the reals), $\varepsilon_1 - \varepsilon_2 := \varepsilon_1 +$ neg ε_2, $\varepsilon_1/\varepsilon_2 := \varepsilon_1 *$ inv ε_2, and $\varepsilon_1 < \varepsilon_2 < \varepsilon_3 :=$ if $\varepsilon_1 < \varepsilon_2$ then $\varepsilon_2 < \varepsilon_3$ else false. Finally, free variables, capture-avoiding substitution, and typing contexts (Γ) are defined in the usual way. The probability context Υ is used to additionally keep track of the distributions that random variables are bound to. When we use Υ in places Γ is expected, the understanding is that the extra information carried by Υ is ignored.

2.3.2 Examples of expressible distributions

With just random, return, and bind, we can already construct a wide variety of distributions we might care to use in practice. Though we do not have a formal proof of this expressivity, existing work on sampling suggests that this is the case. Non-uniform random variate generation is concerned with generating samples from arbitrary distributions using only samples from Uniform(0,1) [18]. We can see the connection with our language if we view the constructs by a sampling analogy, which emphasizes understanding a distribution by its generating process: the phrase var $x \sim e_1$ in e_2 samples a value x from the sampling

function e_1, which is used to create a new sampling function e_2; random samples from the Uniform(0,1) distribution; return ε always returns ε as its sample. For instance, the *standard normal distribution* can be defined in our language using the Box-Muller sampling method:

$$\begin{aligned}\mathsf{std\_normal} \;:=\; & \mathsf{var}\ u \sim \mathsf{random}\ \mathsf{in}\ \mathsf{var}\ v \sim \mathsf{random}\ \mathsf{in} \\ & \quad \mathsf{return}\ (\mathsf{sqrt}(-2 * \mathsf{log}\ u) * \mathsf{cos}(2 * \pi * v))\end{aligned}$$

where $\mathsf{sqrt}\ \varepsilon := \mathsf{exp}\ ((1/2) * \mathsf{log}\ \varepsilon)$. In particular, our language is amenable to *inverse transform sampling*. Likewise, we can express other common continuous distributions:

$$\begin{aligned}&\mathsf{uniform}\ \varepsilon_1\ \varepsilon_2\ := && \mathsf{std\_logistic}\ := \\ &\ \mathsf{var}\ u \sim \mathsf{random}\ \mathsf{in} && \ \mathsf{var}\ u \sim \mathsf{random}\ \mathsf{in} \\ &\ \ \mathsf{return}\ ((\varepsilon_2 - \varepsilon_1) * u + \varepsilon_1) && \ \ \mathsf{return}\ (\mathsf{log}\ (1/u - 1)) \\ &\mathsf{normal}\ \varepsilon_1\ \varepsilon_2\ := && \mathsf{std\_exponential}\ := \\ &\ \mathsf{var}\ x \sim \mathsf{std\_normal}\ \mathsf{in} && \ \mathsf{var}\ u \sim \mathsf{random}\ \mathsf{in} \\ &\ \ \mathsf{return}\ (\varepsilon_2 * x + \varepsilon_1) && \ \ \mathsf{return}\ (-\mathsf{log}\ u)\end{aligned}$$

These are the Uniform(a,b), standard logistic, standard exponential, and Normal(μ, σ) distributions. We intentionally parameterize the normal distribution by its standard deviation instead of its variance, for simplicity. We define it as a transformation of a standard normal random variable and require $\varepsilon_2 > 0$. We can also express discrete distributions, such as the coin flip distribution,

$$\mathsf{flip}\ \varepsilon := \mathsf{var}\ u \sim \mathsf{random}\ \mathsf{in}\ \mathsf{return}\ (u < \varepsilon),$$

which takes the value true with probability ε. This is equivalent to the Bernoulli distribution. In fact, we can express any distribution with finitely many outcomes:

$$\begin{aligned}&\mathsf{var}\ u \sim \mathsf{random}\ \mathsf{in} \\ &\quad \mathsf{return}\ (\mathsf{if}\ u < 1/3\ \mathsf{then}\ 10\ \mathsf{else\ if}\ u < 2/3\ \mathsf{then}\ 20\ \mathsf{else}\ 30)\end{aligned}$$

Admittedly, a more satisfying definition would be possible if we had lists in the language. The reason we do not is that, although others have addressed recursion and iteration in the context of defining probability distributions [34], we have not yet fully reconciled recursion with PDFs. The absence of recursion also means that we do not support distributions in

the style of *rejection sampling* methods, which resample values until a stopping criterion is met. Furthermore, we do not elegantly support *infinite* discrete distributions in the core language, many of which are naturally described using recursion. However, in Section 2.5 we describe how to add special support for any distribution with a known PDF.

We also define some higher-order concepts. The following functions are used to create joint distributions and mixture models:

$$\begin{aligned}
&\mathsf{join}\ e_1\ e_2 := \\
&\quad \mathsf{var}\ x_1 \sim e_1\ \mathsf{in} \\
&\quad\quad \mathsf{var}\ x_2 \sim e_2\ \mathsf{in} \\
&\quad\quad\quad \mathsf{return}\ (x_1, x_2)
\end{aligned}
\qquad\qquad
\begin{aligned}
&\mathsf{mix}\ \varepsilon\ e_1\ e_2 := \\
&\quad \mathsf{var}\ z \sim \mathsf{flip}\ \varepsilon\ \mathsf{in} \\
&\quad\quad \mathsf{var}\ x_1 \sim e_1\ \mathsf{in} \\
&\quad\quad\quad \mathsf{var}\ x_2 \sim e_2\ \mathsf{in} \\
&\quad\quad\quad\quad \mathsf{return}\ (\mathsf{if}\ z\ \mathsf{then}\ x_1\ \mathsf{else}\ x_2).
\end{aligned}$$

The mixture model is created by flipping a coin with the specified probability to determine which component distribution to sample from. For instance, a simple mixture-of-Gaussians is given by mix (1/2) std_normal std_normal. We have defined discrete and continuous distributions, and now we can use join to define non-trivial hybrid distributions as well, such as join (flip (1/2)) random, which has type dist (bool × R). Essentially, if-expressions enable mixture models and tuples enable joint models. These two concepts are special cases of *hierarchical models*, which are models that are defined in stages. Distributions defined using nested instances of bind correspond to hierarchical models.

These examples are all AC, but we can also express non-AC distributions, such as the example from Section 2.2, written jumpy := mix (1/2) (return 0) std_normal. In our language, jumpy will successfully type check as a distribution, but the program pdf jumpy will be rejected—as it should be—because jumpy is not absolutely continuous. The ability to represent non-AC distributions, even though they cannot be used in programs, is in anticipation of future language features such as expectation and sampling, which *can* be used with non-AC distributions.

2.3.3 Measure theory preliminaries

Measure theory is the basis of modern probability theory and unifies the concepts of discrete and continuous probability distributions. It is a precise way of defining the notion of volume. We develop our formalization within this framework. We give only a brief overview of the necessary concepts; details are available in [47].

Basics Let A be a set we wish to measure. A *σ-algebra* $\mathcal{M}$ on A is a subset of the powerset $\mathcal{P}(A)$ that contains A and is closed under complement and countable union. The pair $(A, \mathcal{M})$ is a *measurable space*. A subset X of A is *$\mathcal{M}$-measurable* if $X \in \mathcal{M}$. In the context of probability, A is the set of outcomes and $\mathcal{M}$ is the set of events. For a function $f : A \to B$, the *f-image* of a subset X of A, written $f[X]$, denotes the set $\{f(x) \mid x \in A\}$, and the *f-preimage* of a subset Y of B, written $f^{-1}[Y]$, denotes the set $\{x \in A \mid f(x) \in Y\}$. When f is on measurable spaces, we say f is *$(\mathcal{M}_A, \mathcal{M}_B)$-measurable* when the f-preimage of any $\mathcal{M}_B$-measurable set is $\mathcal{M}_A$-measurable. Measurable functions are closed under composition. We drop the prefix and say *measurable* (for functions or sets) when it is clear what the σ-algebras are. The σ-algebra machinery is needed to ensure a consistent theory; there are spaces which contain pathological sets that violate intuition about volume, *e.g.* the Banach-Tarski "doubling ball" paradox. Measure theory sidesteps these issues by preferring measurable sets and functions as much as possible. When A is countable, no such problems arise, and we can always take $\mathcal{P}(A)$ for $\mathcal{M}$.

Measures A nonnegative function $\mu : \mathcal{M} \to \mathbb{R}^+ \cup \{\infty\}$ is a *measure* if $\mu(\emptyset) = 0$, $\mu(X) \geq 0$ for all X in $\mathcal{M}$, and $\mu(\bigcup_{i=1}^{\infty} X_i) = \sum_{i=1}^{\infty} \mu(X_i)$ for all sequences of mutually disjoint X_i (*countable additivity*). The triple $(A, \mathcal{M}, \mu)$ is a *measure space*. If additionally $\mu(A) = 1$ then μ is a *probability measure* (conventionally written $\mathbb{P}$), and the triple is a *probability space*. We use the terms *probability measure*, *probability distribution* and *distribution* interchangeably. We use $\mathfrak{C}$ to denote the counting measure, which uses the number of elements of a set as the set's measure. We use $\mathfrak{L}$ to denote the Lebesgue measure on $\mathbb{R}$, which assigns the length $|b - a|$ to an open interval (a, b); the sizes of other sets can be understood by complements

and countable unions of intervals. The *product measure* $\mu_A \otimes \mu_B$ of two measures μ_A and μ_B on measurable spaces $(A, \mathcal{M}_A)$ and $(B, \mathcal{M}_B)$ is the measure μ, on $A \times B$ and the product σ-algebra $\mathcal{M}_A \otimes \mathcal{M}_B$, such that

$$\mu(X \times Y) = \mu_A(X) \cdot \mu_B(Y)$$

for $X \in \mathcal{M}_A$ and $Y \in \mathcal{M}_B$. The measure is unique when μ_A and μ_B are σ-finite. The σ-finiteness condition is a technical condition that is satisfied by all measures we will consider in this paper and requires that the space can be covered by a countable number of pieces of finite measure.

Null sets A measurable set X is *μ-null* if $\mu(X) = 0$; X is said to have μ-measure zero. The empty set is always null, the only $\mathfrak{C}$-null set is the empty set, and all countable subsets of $\mathbb{R}$ are $\mathfrak{L}$-null. A propositional function holds *μ-almost everywhere (μ-a.e.)* if the set of elements for which the proposition does not hold is μ-null. For instance, two functions of type $\mathbb{R} \to \mathbb{R}$ are equal $\mathfrak{L}$-almost everywhere if they differ at only a countable number of points. A measure space $(A, \mathcal{M}, \mu)$ is *complete* if all subsets of any μ-null set are $\mathcal{M}$-measurable. *Completion* is an operation that takes any measure space $(A, \mathcal{M}, \mu)$ and produces an "equivalent" complete measure space $(A, \mathcal{M}', \mu')$ such that $\mu'(X) = \mu(X)$ for $X \in \mathcal{M}$. Null sets are ubiquitous in measure theory, so it will be handy to work in spaces that support null sets as much as possible. Thus, completion makes measure spaces nicer to work with. The n-dimensional Lebesgue measure $\mathfrak{L}^n$ is the n-fold completed product of $\mathfrak{L}$. For measures μ and ν on a measurable space, ν is *absolutely continuous* with respect to μ if each μ-null set is also ν-null.

Integration A fundamental operation involving measures is the *abstract integral*, a generalization of the Riemann integral that avoids some of its deficiencies. The abstract integral of a measurable function $f : A \to \mathbb{R}$ w.r.t. a measure μ on A is written $\int f \, d\mu$. The integral is always defined for nonnegative f. The integral for arbitrary f is defined in terms of the positive and negative parts of f and may not exist; if it *does* we say f is μ-integrable. We write $\int_X f \, d\mu$ as shorthand for $\int \lambda x \,.\, \mathbf{1}_X(x) \cdot f(x) \, d\mu$, which restricts the integral to the

subset X. We write $\mathbf{1}_X$ for the indicator function on X. *Expectation* refers to abstract integration w.r.t. a distribution. The abstract integral satisfies $\mu(X) = \int \mathbf{1}_X \, d\mu$ for all measurable X. In terms of probability, it says that the probability of X is the expectation of $\mathbf{1}_X$. Another consequence is that null sets cannot affect integration: two functions that are equal μ-a.e. give the same results under integration w.r.t. μ. Abstract integration w.r.t. $\mathfrak{C}$ and $\mathfrak{L}$ is ordinary (possibly infinite) summation and the ordinary Lebesgue integral, respectively. The Lebesgue integral agrees with the Riemann integral on Riemann-integrable functions.

Measurability Ordinarily, to conclude that a distribution such as $\mathsf{var}\ x \sim e\ \mathsf{in}\ \mathsf{return}\ (f\ x)$ is well-formed, we are obligated to verify that f is a measurable function. However, non-measurable sets and functions are actually quite pathological and constructing them requires the Axiom of Choice [62]. None of the constructs in our language are as powerful as the Axiom of Choice (though we do not have a formal proof of this), thus all constructible expressions represent measurable functions. This discharges the obligation, and we do not make any further mention of checking for measurability.

Stocked spaces For most applications, we often have a standard idea of how spaces are measured. We now formalize this practice. A space A is a *stocked space* if it comes equipped with a complete measure space $(A, \overline{\mathcal{M}}_A, \overline{\mu}_A)$, which is the *stock measure space* of A. We call $\overline{\mathcal{M}}_A$ the *stock σ-algebra* of A and $\overline{\mu}_A$ the *stock measure* of A. The abstract integral w.r.t. $\overline{\mu}_A$ is the *stock integral* of A. We define stock measure spaces for the spaces $\mathbb{B} = \{\text{true}, \text{false}\}$, $\mathbb{Z}$, $\mathbb{R}$, and product spaces between stocked spaces as follows:

$$(\overline{\mathcal{M}}_{\mathbb{B}}, \overline{\mu}_{\mathbb{B}}) = (\mathcal{P}(\mathbb{B}), \mathfrak{C})$$
$$(\overline{\mathcal{M}}_{\mathbb{Z}}, \overline{\mu}_{\mathbb{Z}}) = (\mathcal{P}(\mathbb{Z}), \mathfrak{C})$$
$$(\overline{\mathcal{M}}_{\mathbb{R}}, \overline{\mu}_{\mathbb{R}}) = (\text{the } \mathfrak{L}\text{-measurable sets}, \mathfrak{L})$$
$$(\overline{\mathcal{M}}_{A\times B}, \overline{\mu}_{A\times B}) = \text{completion}(\overline{\mathcal{M}}_A \otimes \overline{\mathcal{M}}_B, \overline{\mu}_A \otimes \overline{\mu}_B).$$

This definition matches what is used in practice: *e.g.* $\mathfrak{C}$ becomes the measure for countable spaces, and $\mathfrak{L}^n$ becomes the measure for $\mathbb{R}^n$. For the rest of the paper, we assume spaces

$$\frac{}{\Gamma \vdash \mathsf{random} : \mathsf{dist}\ \mathsf{R}}\ \textsc{t-rand} \qquad \frac{\Gamma \vdash \varepsilon : \tau}{\Gamma \vdash \mathsf{return}\ \varepsilon : \mathsf{dist}\ \tau}\ \textsc{t-ret}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{dist}\ \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \mathsf{dist}\ \tau_2}{\Gamma \vdash \mathsf{var}\ x \sim e_1\ \mathsf{in}\ e_2 : \mathsf{dist}\ \tau_2}\ \textsc{t-bind}$$

Figure 3: Standard monadic typing rules for distributions.

are stocked, unless explicitly noted otherwise. We say that a distribution on A is AC if it is AC with respect to $\overline{\mu}_A$.

Densities A function f is a PDF of a distribution $\mathbb{P}$ on A if $\mathbb{P}(X) = \int_X f \; d\overline{\mu}_A$ for all measurable X. Expectation can be written using the PDF:

$$\int g \; d\mathbb{P} = \int \lambda x \,.\, g(x) \cdot f(x) \; d\overline{\mu}_A.$$

A *joint* PDF is the PDF of a *joint distribution*, which is simply a distribution on a product space. We later use the fact that the joint PDF f of a model such as $x_1 \sim \mathbb{P}_1, x_2 \sim \mathbb{P}_2(\,\cdot\,; x_1)$ can be written as the product of the individual (parameterized) PDFs: $f(x_1, x_2) = f_1(x_1) \cdot f_2(x_2; x_1)$.

2.3.4 Type system and semantics for distributions

We now discuss the type system and semantics for syntactic categories besides programs. The type system for expressions is ordinary. We assume an external mechanism for enforcing the preconditions necessary to ensure totality of functions, such as an automated theorem prover or the programmer themself. For instance, log must be applied to only positive real numbers. Distributions obey standard monadic typing rules (Figure 3). The "random variables" introduced by bind are really just normal variables and are typed as such; calling them random variables is a reminder about the role they play. The typing rules ensure that random variables are never used outside a probabilistic context.

We give our language a semantics based in classical mathematics with total functions. Base types have the usual meaning. The denotation of dist τ is the set of distributions

possible on τ:

$$\mathcal{T}[\![\mathsf{dist}\ \tau]\!] = \{\mathbb{P} \mid (\mathcal{T}[\![\tau]\!], \overline{\mathcal{M}}_\tau, \mathbb{P}) \text{ is a probability space}\}.$$

We overload stock measure space notation for types; thus, $\overline{\mathcal{M}}_\tau$ and $\overline{\mu}_\tau$ are shorthand for $\overline{\mathcal{M}}_{\mathcal{T}[\![\tau]\!]}$ and $\overline{\mu}_{\mathcal{T}[\![\tau]\!]}$. Let $\mathcal{E}[\![e]\!]\rho$ be the denotation of a distribution e under the environment ρ, also overloaded for expressions ε. Expressions have the semantics of their corresponding forms from classical mathematics. As stated before, random is the Uniform(0,1) distribution

$$\mathcal{E}[\![\mathsf{random}]\!]\rho = \lambda X \,.\, \mathfrak{L}(X \cap [0,1]),$$

which says that the probability of an event X is its "interval size" on [0,1]. Return is the point mass distribution

$$\mathcal{E}[\![\mathsf{return}\ \varepsilon]\!]\rho = \lambda X \,.\, \mathbf{1}_X(\mathcal{E}[\![\varepsilon]\!]\rho),$$

which gives an event X probability 1 as long as it includes the outcome ε. Bind expresses the Law of Total Probability,

$$\mathcal{E}[\![\mathsf{var}\ x \sim e_1\ \mathsf{in}\ e_2]\!]\rho = \lambda Y \,.\, \int \lambda x' \,.\, f(x')(Y)\ d\mathbb{P},$$

where $f(x') = \mathcal{E}[\![e_2]\!](\rho\{x \mapsto x'\})$ and $\mathbb{P} = \mathcal{E}[\![e_1]\!]\rho$. The family of distributions e_2 is parameterized by the variable x, in essence. The probability of an event Y is the "average opinion" (the $\mathbb{P}$-expectation) of what each member of the family thinks is the probability of Y. The integral exists because it is the expectation of a bounded function.

2.4 Type system and semantics for programs

The program $\mathsf{pdf}\ e$ is well-formed if the distribution e permits a PDF. The following theorem gives us a sufficient condition.

Theorem 2.4.1 (Radon-Nikodym)**.** *For any two σ-finite measures μ and ν on the same measurable space such that ν is absolutely continuous w.r.t. μ, there is a function f such that $\nu(X) = \int_X f\ d\mu$.*

We call f a *Radon-Nikodym derivative* of ν with respect to μ, denoted $d\nu/d\mu$; pdf corresponds to the Radon-Nikodym operator. The condition is also necessary: given a

satisfying f, ν is (trivially) AC. All stock measures we define and all distributions are σ-finite, so for our purposes absolute continuity is equivalent to possessing a PDF. Though not necessarily unique, Radon-Nikodym derivatives are equal μ-almost everywhere. When μ is the counting measure, the Radon-Nikodym derivative is a PMF. For hybrid spaces, it is a function which must be summed along one dimension and integrated along the other to obtain quantities interpretable as probabilities. Radon-Nikodym derivatives unify PMFs, PDFs and hybrids of the two. For this reason, we refer to all of these as PDFs. We use "PMF" when we want to emphasize its discrete nature.

Defining a type system for absolute continuity in terms of the straightforward induction on distribution terms proves unwieldy. Suppose we want to check if the distribution in Equation 2 is AC; we must verify that the probability of any $\mathfrak{L}^2$-null set Z is zero. A straightforward induction leads us to trying to show

$$\mathbb{P}(\{x \mid \left(\begin{array}{c} \mathsf{var}\ y \sim \mathsf{random}\ \mathsf{in} \\ \mathsf{return}\ (x, y) \end{array} \right)(Z) \neq 0\}) = 0$$

where $\mathbb{P}$ is the Uniform(0,1) distribution, and we have abused notation slightly by mingling object language syntax with ordinary mathematics. This states that the body of the outermost bind assigns Z probability zero, $\mathbb{P}$-almost always. It is unclear how to proceed from here or how to remove concepts like null sets from the mechanization. We take an alternate approach based on the insight that we can reason about a distribution by examining how it transforms other distributions. Our approach, and outline of the following subsections, is as follows:

- We introduce the new notion of a *non-nullifying* function and prove a transformation theorem stating that when a random variable is transformed, the output distribution is AC if the input distribution is AC and the transformation is non-nullifying. We also prove some results about non-nullifying functions.

- We define the *random variables transform* of any distribution written in our language and show that for a large class of distributions the transformation theorem is applicable.

- We present a type system which defines absolute continuity of a distribution in terms of whether its RV transform is non-nullifying. As implementors, we have found it easier to come up with the rules for non-nullifying functions.

Measure-theoretic concepts like σ-algebras, null sets, and the Lebesgue measure, while present in the metatheory, do not need to be operationalized for implementing the type checker. Also, due to the measure-theoretic foundation, we correctly handle cases that are not typically explained, such as PDFs on hybrid spaces. We conclude the section with the semantics of programs.

2.4.1 Absolute continuity and non-nullifying functions

A function $h : A \to B$ is *non-nullifying* if the h-preimage of each $\overline{\mu}_B$-null set is $\overline{\mu}_A$-null; preimages of null sets are always null, and forward images of non-null sets are always non-null. A function that fails to be non-nullifying is called *nullifying*. The next theorem establishes the link between absolute continuity and non-nullity.

Theorem 2.4.2 (Transformation)**.** *For a function $h : A \to B$ and an* AC *distribution $\mathbb{P}$ on A, the distribution*

$$\mathbb{Q}(Y) = \mathbb{P}(h^{-1}[Y]) \tag{3}$$

on B is AC *if h is non-nullifying.*

Proof. Let Y be a $\overline{\mu}_B$-null set. By the non-nullity of h, the set $h^{-1}[Y]$ is $\overline{\mu}_A$-null. By the absolute continuity of $\mathbb{P}$, we have $\mathbb{P}(h^{-1}[Y]) = 0$, implying that Y is also $\mathbb{Q}$-null. □

This style of defining $\mathbb{Q}$ may seem odd, but it actually underlies the use of random variables as a modeling language. For instance, the model $x \sim \mathbb{P},\ y = h(x)$ exhibits the relationship $\mathbb{Q}(Y) = \mathbb{P}(h^{-1}[Y])$, where $\mathbb{Q}$ is the distribution of y. In general, the reverse direction does not hold; h can be nullifying even if $\mathbb{Q}$ is AC. This happens when h has nullifying behavior only in regions of the space where $\mathbb{P}$ is assigning zero probability. This will be a source of incompleteness in the type system.

Lemma 2.4.3 (Discrete domain)**.** *A function $h : A \to \mathbb{R}$ is nullifying if A is non-empty and countable.*

Proof. Let x be an element of A. The set $\{x\}$ has positive counting measure while its h-image, which is a singleton set, is $\mathfrak{L}$-null. □

This implies R_of_Z is nullifying, meaning that when we view an integer random variable as a real random variable, it loses its ability to have a PDF. This is desirable behavior; different spaces have different ideas of what it means to be a PDF. We would not want to mark an integer random variable as AC and later attempt to integrate its PMF in a context expecting a real random variable.

Lemma 2.4.4 (Discrete codomain)**.** *A function $h : A \to B$ is non-nullifying if B is countable.*

Proof. The h-preimage of the empty set (the only $\mathfrak{C}$-null set) is the empty set, which is always null. □

This reasoning corroborates the fact that distributions on countable spaces always have a PMF.

Lemma 2.4.5 (Interval)**.** *A function $h : \mathbb{R} \to \mathbb{R}$ is nullifying if it is constant on any interval.*

Proof. Let h be constant on (a, b); (a, b) is not $\mathfrak{L}$-null, but its h-image (a singleton set) is $\mathfrak{L}$-null. □

One way to visualize how this leads to a non-AC distribution is to observe that the transformation h takes all the probability mass along (a, b) and non-smoothly concentrates it onto a single point in the target space.

Lemma 2.4.6 (Inverse)**.** *An invertible function $h : \mathbb{R} \to \mathbb{R}$ is non-nullifying if its inverse h^{-1} is an* absolutely continuous function.

Proof. We have discussed absolute continuity of measures; the absolute continuity of functions is a related idea. It is a stronger notion than continuity and uniform continuity. Absolutely continuous functions are well behaved in many ways; in particular, the images of null sets are also null sets. Coupled with the fact that an h-preimage is an h^{-1}-image, this proves the claim. More details on absolutely continuous functions can be found in [47]. □

This result shows that log, exp, and non-constant linear functions are non-nullifying. We believe the idea can be extended without much difficulty to show that functions with a countable number of invertible pieces, such as the trigonometric functions and non-constant polynomials, are also non-nullifying.

Lemma 2.4.7 (Piecewise)**.** *For functions $c : A \to \mathbb{B}$ and $f, g, h : A \to B$, where $h(x) =$ if $c(x)$ then $f(x)$ else $g(x)$, h is non-nullifying if f and g are non-nullifying.*

Proof. Let Y be a $\overline{\mu}_B$-null set. The set $h^{-1}[Y]$ is a subset of $f^{-1}[Y] \cup g^{-1}[Y]$ and is thus $\overline{\mu}_A$-null, by non-nullity of f and g, and the countable additivity and completeness of $\overline{\mu}_A$. □

Lemma 2.4.8 (Composition)**.** *The set of non-nullifying functions is closed under function composition.*

Proof. Let $f : A \to B$ and $g : B \to C$ be non-nullifying functions and let $h = g \circ f$. The h-preimage of a $\overline{\mu}_C$-null set Z is given by $h^{-1}[Z] = f^{-1}[g^{-1}[Z]]$, and is thus $\overline{\mu}_A$-null, by the non-nullity of f and g. □

Lemma 2.4.9 (Projection)**.** *The function $h(x, y) = x$ of type $A \times B \to A$ is non-nullifying.*

Proof. Let X be a $\overline{\mu}_A$-null set. Its h-preimage is $X \times B$. By the properties of product measure, we have that

$$\overline{\mu}_{A\times B}(X \times B) = \overline{\mu}_A(X) \cdot \overline{\mu}_B(B) = 0 \cdot \overline{\mu}_B(B) = 0.$$

Even if $\overline{\mu}_B(B) = \infty$, the measure-theoretic definition of multiplication on extended non-negative reals defines $0 \cdot \infty = 0$. □

Along these lines, we can show that returning a permutation of a subset of tuple components is also a non-nullifying function. The last two results permit us to ignore uninvolved arguments when reasoning about the non-nullity of the body of a function.

2.4.2 Distributions and RV transforms

A large class of distributions in our language can be understood by Equation 3. From the syntax we know that a distribution e must take the form of zero or more nested binds terminating in a body that is either random or return ε. We focus on the latter, non-trivial case. The expression ε represents a transformation of the random variables x_i introduced by the binds. The function $\lambda(x_1, ..., x_n) \,.\, \varepsilon$ is the *random variables transform (RV transform)* of the distribution e, where we use tuple pattern matching shorthand to name the components of a tuple argument. The correspondence between distributions in our language and Theorem 2.4.2 is as follows: let $\mathbb{Q}$ be the denotation of e, let h be the RV transform of e, and let $\mathbb{P}$ be the joint distribution of the random variables introduced on the spine of e. The class of distributions for which the theorem is applicable is given by the set of distributions for which each e_i is *parametrically* AC w.r.t. the random variables preceding it, where e_i is the distribution corresponding to x_i. In other words, the distribution for e_i must be AC while treating free occurrences of $x_1, ..., x_{i-1}$ as fixed, unknown constants. This ensures that the joint distribution is also AC; the joint PDF can be written as the product of the individual parameterized PDFs. This is a commonly used (implicit) assumption in practice. For example, the distribution

$$\mathsf{var}\ u \sim \mathsf{random}\ \mathsf{in}\ \mathsf{var}\ z \sim \mathsf{flip}\ u\ \mathsf{in}\ \mathsf{return}\ (u + \langle z \rangle)$$

has the RV transform $\lambda(u, z) \,.\, u + \langle z \rangle$, which has type $\mathbb{R} \times \mathbb{B} \to \mathbb{R}$ and is transforming the joint distribution of random and $e_2 := \mathsf{flip}\ u$. The variable u appears free in e_2, making e_2 parametric in u; the restriction requires that e_2 is AC for all possible values of u, which is the case here. Two extensionally equivalent distributions may have different RV transforms and spines because of intensionally different representations. To show that this choice of $\mathbb{P}$, $\mathbb{Q}$, and h satisfies Equation 3, we appeal to the semantics of distributions (defined in Section 2.3.4). Consider the general case $e := \overline{\mathsf{var}\ x_i \ \sim \ e_i\ \mathsf{in}}\ \mathsf{return}\ \varepsilon$, where we have used

$$\frac{}{\Upsilon;\Lambda \vdash \mathsf{random}\ \text{AC}}\ \text{AC-RAND} \qquad \frac{\Upsilon;\Lambda \vdash \varepsilon\ \text{NN}}{\Upsilon;\Lambda \vdash \mathsf{return}\ \varepsilon\ \text{AC}}\ \text{AC-RET}$$

$$\frac{\Upsilon;\emptyset \vdash e_1\ \text{AC} \qquad \Upsilon \vdash e_1 : \mathsf{dist}\ \tau \qquad \Upsilon, x : \tau \sim e_1; \Lambda, x \vdash e_2\ \text{AC}}{\Upsilon;\Lambda \vdash \mathsf{var}\ x \sim e_1\ \mathsf{in}\ e_2\ \text{AC}}\ \text{AC-BIND}$$

Figure 4: The absolute continuity judgment, $\Upsilon;\Lambda \vdash e\ \text{AC}$.

the bar as shorthand for nested binds. The denotation $\mathbb{Q}$ of e under an environment ρ is given by

$$\mathbb{Q}(Y) = \overline{\int d\mathbb{P}_i\ \lambda x_i' \centerdot\ \mathbf{1}_Y(\mathcal{E}[\![\varepsilon]\!]\rho\overline{\{x_i \mapsto x_i'\}})}$$

where $\mathbb{P}_i$ is the denotation of e_i (extending ρ as necessary) and we have again used the bar notation, to denote iterated expectation and the repeated extension of the environment ρ with variable mappings. We can now rewrite the expectations to use their corresponding PDFs f_i and then replace the iterated integrals with a single product integral using their joint PDF f:

$$\begin{aligned}
\mathbb{Q}(Y) &= \overline{\int d\overline{\mu}_{\tau_i} \lambda x_i' \centerdot f_i(x_i'; x_1', ..., x_{i-1}') \cdot \mathbf{1}_Y(\mathcal{E}[\![\varepsilon]\!]\rho\overline{\{x_i \mapsto x_i'\}})} \\
&= \int d\overline{\mu}_\tau\ \lambda \mathbf{x} \centerdot\ f(\mathbf{x}) \cdot \mathbf{1}_Y(h(\mathbf{x})) \\
&= \int d\mathbb{P}\ \lambda \mathbf{x} \centerdot\ \mathbf{1}_Y(h(\mathbf{x})) \\
&= \mathbb{P}(\{\mathbf{x} \mid h(\mathbf{x}) \in Y\})\ =\ \mathbb{P}(h^{-1}[Y])
\end{aligned}$$

where $\mathbf{x} = (x_1', ..., x_n')$, $h(\mathbf{x}) = \mathcal{E}[\![\varepsilon]\!]\rho\overline{\{x_i \mapsto \mathbf{x}_i\}}$, τ_i is the type of each x_i, and τ is their product. We have also used the fact that the expectation of the indicator function on a set is the probability of that set (the set here is $\{\mathbf{x} \mid h(\mathbf{x}) \in Y\}$, not Y). Replacing an iterated integral with a product integral is not always legal but is possible here because the integral is of a nonnegative function w.r.t. independent measures (see Tonelli's theorem, [47]).

2.4.3 Type system for programs

All judgments are defined modulo α-conversion. A program $\mathsf{pdf}\ e$ is well-formed if e is an AC distribution ($\emptyset \vdash e : \mathsf{dist}\ \tau$ holds for some τ and $\emptyset;\emptyset \vdash e\ \text{AC}$ holds). If the judgment

$$\frac{x \in \Lambda}{\Upsilon; \Lambda \vdash x \text{ NN}} \text{ NN-VAR} \qquad \frac{\Upsilon \vdash \varepsilon : \tau \qquad \tau \text{ countable}}{\Upsilon; \Lambda \vdash \varepsilon \text{ NN}} \text{ NN-COUNT}$$

$$\frac{\Upsilon; \Lambda \vdash \varepsilon \text{ NN} \qquad op \in \{\mathsf{neg}, \mathsf{inv}, \mathsf{log}, \mathsf{exp}, \mathsf{sin}, \mathsf{cos}, \mathsf{tan}\}}{\Upsilon; \Lambda \vdash op\ \varepsilon \text{ NN}} \text{ NN-OP}$$

$$\frac{\Upsilon; \Lambda \vdash \varepsilon_1 \text{ NN} \qquad \Upsilon; \Lambda \vdash \varepsilon_2 \text{ NN}}{\Upsilon; \Lambda \vdash \mathsf{if}\ \varepsilon\ \mathsf{then}\ \varepsilon_1\ \mathsf{else}\ \varepsilon_2 \text{ NN}} \text{ NN-IF}$$

$$\frac{\Upsilon; \Lambda \vdash \varepsilon \text{ NN} \qquad op \in \{\mathsf{fst}, \mathsf{snd}\}}{\Upsilon; \Lambda \vdash op\ \varepsilon \text{ NN}} \text{ NN-PROJ}$$

$$\frac{\Upsilon; \Lambda \vdash \varepsilon_1 \perp \varepsilon_2 \qquad \Upsilon; \Lambda \vdash \varepsilon_1 \text{ NN} \qquad \Upsilon; \Lambda \vdash \varepsilon_2 \text{ NN}}{\Upsilon; \Lambda \vdash (\varepsilon_1, \varepsilon_2) \text{ NN}} \text{ NN-PAIR}$$

$$\frac{x_i \in \Lambda \qquad x_1, ..., x_n \text{ are distinct}}{\Upsilon; \Lambda \vdash (x_1, ..., x_n) \text{ NN}} \text{ NN-VARS}$$

$$\frac{\Upsilon; \Lambda \vdash (\varepsilon_1, \varepsilon_2) \text{ NN}}{\Upsilon; \Lambda \vdash \varepsilon_1 + \varepsilon_2 \text{ NN}} \text{ NN-PLUS}$$

$$\frac{FV(\varepsilon_2) \cap \Lambda = \emptyset \qquad \Upsilon; \Lambda \vdash \varepsilon_1 \text{ NN}}{\Upsilon; \Lambda \vdash \varepsilon_1 + \varepsilon_2 \text{ NN}} \text{ NN-LINEAR}$$

$$\frac{\Upsilon; \Lambda \vdash (\varepsilon_1, \varepsilon_2) \text{ NN}}{\Upsilon; \Lambda \vdash \varepsilon_1 * \varepsilon_2 \text{ NN}} \text{ NN-MULT} \qquad \frac{l \neq 0 \quad \Upsilon; \Lambda \vdash \varepsilon \text{ NN}}{\Upsilon; \Lambda \vdash l * \varepsilon \text{ NN}} \text{ NN-SCALE}$$

Figure 5: The non-nullity judgment, $\Upsilon; \Lambda \vdash \varepsilon$ NN.

$\Upsilon; \Lambda \vdash e$ AC (Figure 4) holds then e is an AC distribution under the probability context Υ and the active variable context Λ, where Λ is given by the grammar $\Lambda ::= \emptyset \mid \Lambda, x$. Variables in Λ are currently active and should be understood in a probabilistic sense, while those not in Λ are inactive and should be treated as fixed parameters. The contexts obey the following invariant: Λ is always the "prefix" of Υ, *i.e.* the variables in Λ correspond directly to the n most recent entries added to Υ, where n is the length of Λ. Rule AC-RAND asserts that the Uniform(0,1) distribution is AC. The main action of rules AC-BIND and AC-RETURN is to prepare a call to the non-nullity judgment. For Theorem 2.4.2 to be applicable, a distribution along the spine must be parametrically AC w.r.t. the random variables preceding it; thus, in AC-BIND we check that e_1 is AC without marking any current random variables as active. We reach the body of the RV transform in AC-RETURN. Roughly speaking, Λ (pointing into Υ) and ε correspond to $\mathbb{P}$ and h in Theorem 2.4.2.

Next is the non-nullity judgment (Figure 5). If $\Upsilon; \Lambda \vdash \varepsilon$ NN holds, then ε represents the body of a non-nullifying function under Υ and Λ. The variables in Λ are the arguments to the

RV transform. Throughout this discussion, we implicitly use the composition and projection lemmas (Lemmas 2.4.8 and 2.4.9) to ignore uninvolved arguments during analysis. For example, in rule NN-VAR, we could be analyzing a function with multiple inputs, but we can drop all of them but x, leaving us to analyze the function $\lambda x \,.\, x$, which is trivially non-nullifying. Under the hood, what we are actually doing is representing the original transform as the composition of a function that selects a single components of a tuple with the identity function $\lambda x \,.\, x$. The composition lemma is also the justification for being able to recurse into subexpressions. Rule NN-COUNT is merely an application of Lemma 2.4.4; the types bool, Z and products thereof define the countable types. Note that this covers the cases of $=$, $<$, integer neg, $+$ and $*$, and Boolean and integer literals. Rules NN-OP, NN-IF and NN-PROJ are direct translations of Lemmas 2.4.6, 2.4.7 and 2.4.9. The injection from integers into the reals is nullifying (Lemma 2.4.3), so there is no rule for $\mathsf{R\_of\_Z}$. Rule NN-PAIR expresses the idea that the joint distribution of independent AC distributions is AC. If $\Upsilon; \Lambda \vdash \varepsilon_1 \perp \varepsilon_2$ holds then ε_1 and ε_2 represent independent distributions under Υ and Λ. Its definition is

$$\frac{\Lambda \;\cap\; \mathrm{Anc}(\Upsilon, FV(\varepsilon_1)) \;\cap\; \mathrm{Anc}(\Upsilon, FV(\varepsilon_2)) = \emptyset}{\Upsilon; \Lambda \vdash \varepsilon_1 \perp \varepsilon_2} \text{ INDEP}$$

where $\mathrm{Anc}(\Upsilon, X) = \bigcup_{x \in X} \mathrm{anc}(\Upsilon, x)$. It states that ε_1 and ε_2 must not have any ancestors in common. The function $\mathrm{anc}(\Upsilon, x)$ computes the ancestors of a random variable x. A random variable y is the parent of a random variable x if y appears free in the distribution that x is bound to. Rule NN-VARS corresponds to the corollary of Lemma 2.4.9 that states that you can drop and permute tuple components. The requirement that the variables are distinct is important; the distribution $\mathsf{var}\ u \sim \mathsf{random\ in\ return}\ (u, u)$ is not AC, as we saw in Section 2.2. We have multiple rules for addition because they each capture a different usage of plus. Rule NN-PLUS states that if the formation of the pair $(\varepsilon_1, \varepsilon_2)$ is non-nullifying, then $\varepsilon_1 + \varepsilon_2$ is also non-nullifying because it is the composition of tuple formation with $(+) : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, where the latter is non-nullifying by corollary to Lemma 2.4.6. Rule NN-LINEAR represents the idea of composing with the non-nullifying function $\lambda x \,.\, x + c$, where c is a constant w.r.t. the arguments of the RV transform. There is an analogous

rule for when the constant appears as the left operand. Rules NN-MULT and NN-SCALE are analogous. Note that NN-SCALE is slightly weaker than its counterpart NN-LINEAR, only because it needs to prove that the scaling coefficient is nonzero.

Discussion We believe our type system is sound; the only remaining case to rigorously prove is NN-PAIR. The soundness of the reduction to non-nullity is given by Theorem 2.4.2, and the soundness of the other cases in the non-nullity judgment are covered by the lemmas in Section 2.4.1. Stating the needed lemma for NN-PAIR essentially requires formalizing the idea that the conditional distribution of the second component conditioned on the first component should be AC. In non-nullity terms, the second component should still have a degree of freedom even after fixing the first. Rigorously stating this involves conditional probability, putting it outside the scope of the current work.

There are few sources of incompleteness in our type system. For instance, NN-PAIR conservatively requires ε_1 and ε_2 to be independent. The distribution

$$\mathsf{var}\ x \sim \mathsf{random}\ \mathsf{in}\ \mathsf{var}\ y \sim \mathsf{random}\ \mathsf{in}\ \mathsf{return}\ (\mathsf{exp}\ x, x + y)$$

is AC despite the fact that the tuple components are not independent: even if we know the value of $\mathsf{exp}\ x$, the "residual" stochasticity in the quantity $x + y$ is still AC. The joint PDF is given by multiplying the marginal PDF of the first component by the conditional PDF of the second component conditioned on the first. This is a similar issue as the parametric AC requirement on spine distributions. Formulating this generalization of NN-PAIR is interesting future work. Likewise, NN-IF conservatively requires both branches of an if-expression to be non-nullifying. The distribution

$$\mathsf{var}\ x \sim \mathsf{std\_normal}\ \mathsf{in}\ \mathsf{return}(\mathsf{if}\ x{<}0\ \mathsf{then}\ \mathsf{min}\ x\ 0\ \mathsf{else}\ \mathsf{max}\ x\ 0)$$

is not accepted as AC because both branches ($\lambda x\,.\,\mathsf{min}\ x\ 0$ and $\lambda x\,.\,\mathsf{max}\ x\ 0$) are nullifying, even though the distribution is extensionally equivalent to $\mathsf{var}\ x \sim \mathsf{std\_normal}\ \mathsf{in}\ \mathsf{return}\ x$, which is AC. We define min and max in the usual way, using if. Finally, non-nullity is sufficient but not necessary for absolute continuity to hold. For instance, the RV transform

of

$$\mathsf{var}\ x \sim \mathsf{random}\ \mathsf{in}\ \mathsf{return}\ (\mathsf{if}\ x < 100\ \mathsf{then}\ x\ \mathsf{else}\ 100)$$

is $\lambda x\,.\,\mathsf{if}\ x < 100\ \mathsf{then}\ x\ \mathsf{else}\ 100$, which is nullifying due to the constant portion, thus our type system does not accept this distribution as AC. However, x only takes values on $(0, 1)$, so the second branch is never entered, and thus the distribution is extensionally equivalent to the AC distribution $\mathsf{var}\ x \sim \mathsf{random}\ \mathsf{in}\ \mathsf{return}\ x$.

2.4.4 Semantics of programs

The denotation of a program $\mathsf{pdf}\ e$ is that it is a member of the set of Radon-Nikodym derivatives of the distribution e:

$$[\![\mathsf{pdf}\ e]\!] \in \{f \mid \forall X,\ \mathbb{P}(X) = \int_X f\ d\overline{\mu}_\tau\}$$

where $\mathbb{P} = \mathcal{E}[\![e]\!]\{\}$ is the denotation of e under the empty environment and e has type $\mathsf{dist}\ \tau$. The procedure discussed in the next section calculates a member of this set.

2.5 Calculating density functions

The previous sections have defined a language in which it is possible to express PDFs. Our goal now is to mechanically obtain a usable form of the PDF for a given distribution. But what constitutes a usable form? We are motivated by applications of the PDF and the need to interface with existing software. For instance, we may want to use numerical optimization software to perform MLE, where the PDF appears in the objective function; we may also want to symbolically derive gradient information to improve the search. Or, we may want to use the PDF to calculate an expectation using a numerical integrator. Roughly speaking, we call a term "usable" if we can map it onto the capabilities of existing software in accordance with common practice. For example, the term $\lambda x.\ x+5$ is usable; in practice, real addition is mapped to floating point addition. Likewise, $\int_0^5 x^2\ dx$ is usable; the integral is Riemannian and in a form accepted by computer algebra systems (CAS) and numerical integrators. On the other hand, terms like $\int g\ d\mathbb{P}$ and $d\mathbb{P}/d\mathfrak{L}$ make use of measure-theoretic operations such as abstract integration and the Radon-Nikodym derivative. Current software do not handle

$$
\begin{array}{rl}
\text{Target types} & \sigma ::= \tau \mid \sigma_1 \to \sigma_2 \\
\text{Target terms} & \delta ::= \varepsilon \mid \lambda x : \tau \centerdot \delta \mid \delta_1\ \delta_2 \mid \int \delta \\
\text{Typing} & \dfrac{\Gamma \vdash \delta : \tau \to \mathsf{R}}{\Gamma \vdash \int \delta : \mathsf{R}}\ \textsc{t-int} \\
\text{Semantics} & \mathcal{E}[\![\int \delta\ ]\!]\rho = \int \mathcal{E}[\![\delta]\!]\rho\ \ d\overline{\mu}_\tau
\end{array}
$$

Figure 6: The target language.

these operations (though, progress on mechanizing measure theory has been made [40]). Thus, the basic plan is to eliminate measure-theoretic concepts during PDF calculation. This means the constructs random, return, bind, and pdf should not appear in a PDF term because they involve measure theory, metatheoretically.

It will take some ingenuity to remove the Radon-Nikodym derivative (pdf). It has been shown that the Radon-Nikodym derivative is a non-computable operator: given a distribution, there is no general computable procedure for computing its PDF [28]. The discrete case at least enjoys the fact that the PMF has a straightforward definition in terms of its distribution; if $\mathbb{P}$ is an executable implementation of a discrete distribution, an executable implementation of its PMF $d\mathbb{P}/d\mathfrak{C}$ is given by $\lambda x \centerdot \mathbb{P}(\{x\})$. In general, however, we will need to tackle the calculation of PDFs with a collection of techniques. Our basic approach is as follows. First, we define a target language that defines what constitutes a usable form. Second, we provide a procedure that converts many distributions accepted as AC by our type system into PDFs expressed in the target language. Some RV transforms are mathematically inconvenient, so we will not be able to calculate certain PDFs from scratch; in particular, dependence between random variables makes the general case difficult. However, the design permits modularly adding knowledge about individual distributions with known PDFs, enabling the procedure to calculate PDFs for programs that use these distributions as subcomponents. This allows us to handle many useful cases.

$$\mathsf{random}\ \$\ \delta \mapsto \int \lambda x : \mathsf{R}\,.\ \langle 0 < x < 1 \rangle * \delta\ x$$

$$\mathsf{return}\ \varepsilon\ \$\ \delta \mapsto \delta\ \varepsilon$$

$$\frac{e_2\ \$\ \delta \mapsto \delta' \qquad e_1\ \$\ \lambda x\,.\,\delta' \mapsto \delta''}{\mathsf{var}\ x \sim e_1\ \mathsf{in}\ e_2\ \$\ \delta \mapsto \delta''}$$

Figure 7: The probability compiler, $e\ \$\ \delta \mapsto \delta'$.

2.5.1 The target language

The target language extends expressions with λ-abstraction, application, and the stock integral (Figure 6). We treat functions in a standard way. Notationally, we skip specifying τ in abstractions when the choice of τ is clear. Computing closed-form solutions for integrals is not always feasible or possible, so integrals cannot be completely eliminated from the target language. The integral is well-formed if its integrand is real-valued and summable (a function f is *μ-summable* if $\int f\ d\mu$ is finite). We require users of the target language (compiler writers) to manually ensure summability; this is reasonable for a back-end language. We have verified summability for each use of stock integration in the compilers presented in this section. Although a measure-theoretic concept, stock integration is close enough to the notion of integration used by numerical and symbolic solvers to be useful as a compilation target. Recall, stock integration over $\mathfrak{C}$ and $\mathfrak{L}$ is ordinary summation and Lebesgue integration, respectively. For most applications, Lebesgue integration will coincide with Riemann integration.

2.5.2 The probability compiler

We need to calculate probabilities as a subroutine of PDF calculation. We achieve this by translating distributions into Kozen-style terms [34]. The probability compiler $e\ \$\ \delta \mapsto \delta'$ performs this translation (Figure 7). It takes a distribution e of type $\mathsf{dist}\ \tau$ and a function δ from τ to $[0, 1]$ and returns the expectation of δ w.r.t. e. When δ is the indicator function on a set X, δ' is the e-probability of X. For instance, suppose we want to know the probability that a sample from $\mathsf{flip}\ (3/4)$ is true. We invoke the probability compiler with $e := \mathsf{flip}\ (3/4)$

$$\frac{}{\Upsilon; \Lambda \vdash \mathsf{random} \curvearrowright \lambda x : \mathsf{R} \,.\, \langle 0 < x < 1 \rangle} \text{ P-RAND}$$

$$\frac{\Upsilon \vdash e_1 : \mathsf{dist}\ \tau \qquad \Upsilon, x : \tau \sim e_1; \Lambda, x \vdash e_2 \curvearrowright \delta}{\Upsilon; \Lambda \vdash \mathsf{var}\ x \sim e_1\ \mathsf{in}\ e_2 \curvearrowright \delta} \text{ P-BIND}$$

$$\frac{\Upsilon; \Lambda \vdash \varepsilon \rightsquigarrow \delta}{\Upsilon; \Lambda \vdash \mathsf{return}\ \varepsilon \curvearrowright \delta} \text{ P-RET}$$

Figure 8: The distribution-to-PDF converter, $\Upsilon; \Lambda \vdash e \curvearrowright \delta$.

and $\delta := \lambda z : \mathsf{bool} \,.\, \langle z \rangle$, producing

$$\int \lambda x : \mathsf{R} \,.\, \langle 0 < x < 1 \rangle * (\lambda u \,.\, (\lambda z \,.\, \langle z \rangle)\ (u < 3/4))\ x$$

for δ', which is equivalent to $\int_0^1 \langle x < 3/4 \rangle\ dx = 3/4$, as expected. Likewise, to derive the probability that a standard normal random variable stays within a standard deviation of its mean, we would invoke the probability compiler with $e := \mathsf{std\_normal}$ and $\delta := \lambda x \,.\, \langle -1 < x < 1 \rangle$. Details on how this computes probabilities are given by Kozen and can also be understood by the expectation monad [55]. We also need the judgment $\Upsilon \vdash \varepsilon \,\$\, \delta \mapsto \delta'$, which invokes the probability compiler on the distribution corresponding to the RV transform body ε in the context Υ.

2.5.3 The pdf calculation procedure

We structure the PDF calculation procedure as we did the type system: the judgment on distributions prepares a call to the judgment on RV transforms. The PDF of a well-formed program $\mathsf{pdf}\ e$ is given by the δ satisfying $\emptyset; \emptyset \vdash e \curvearrowright \delta$. The judgment $\Upsilon; \Lambda \vdash e \curvearrowright \delta$ calculates the PDF δ of the distribution e under Υ and Λ (Figure 8). Rule P-RAND gives the PDF of Uniform(0,1): the indicator function on $(0, 1)$. Rules P-RET and P-BIND build the contexts and invoke the next compiler. The real work begins in the judgment $\Upsilon; \Lambda \vdash \varepsilon \rightsquigarrow \delta$, which computes the PDF δ corresponding to the RV transform body ε under Υ and Λ. We present this judgment in two parts, one each for univariate and multivariate transforms. The multivariate transforms must deal with the issue of dependence between inputs or between outputs of the transform.

$$\frac{\Upsilon;\Lambda \vdash \varepsilon \rightsquigarrow \delta}{\Upsilon;\Lambda \vdash \mathsf{log}\ \varepsilon \rightsquigarrow \lambda x : \mathsf{R}\centerdot\ \delta\ (\mathsf{exp}\ x) * \mathsf{exp}\ x}\ \text{P-LOG}$$

$$\frac{\Upsilon;\Lambda \vdash \varepsilon \rightsquigarrow \delta}{\Upsilon;\Lambda \vdash \mathsf{exp}\ \varepsilon \rightsquigarrow \lambda x : \mathsf{R}\centerdot\ \delta\ (\mathsf{log}\ x) * (1/x)}\ \text{P-EXP}$$

$$\frac{FV(\varepsilon_2) \cap \Lambda = \emptyset \qquad \Upsilon;\Lambda \vdash \varepsilon_1 \rightsquigarrow \delta}{\Upsilon;\Lambda \vdash \varepsilon_1 + \varepsilon_2 \rightsquigarrow \lambda x : \mathsf{R}\centerdot\ \delta\ (x - \varepsilon_2)}\ \text{P-LINEAR}$$

$$\frac{\Upsilon;\Lambda \vdash \varepsilon \rightsquigarrow \delta \qquad l > 0}{\Upsilon;\Lambda \vdash l * \varepsilon \rightsquigarrow \lambda x : \mathsf{R}\centerdot\ \delta\ (x/l) * (1/l)}\ \text{P-SCALE}$$

$$\frac{\Upsilon;\Lambda \vdash \varepsilon \rightsquigarrow \delta}{\Upsilon;\Lambda \vdash \mathsf{neg}\ \varepsilon \rightsquigarrow \lambda x : \mathsf{R}\centerdot\ \delta\ (-x)}\ \text{P-NEG}$$

$$\frac{\Upsilon;\Lambda \vdash \varepsilon \rightsquigarrow \delta}{\Upsilon;\Lambda \vdash \mathsf{inv}\ \varepsilon \rightsquigarrow \lambda x : \mathsf{R}\centerdot\ \delta\ (1/x) * (1/(x * x))}\ \text{P-INV}$$

Figure 9: The transform-to-PDF converter, $\Upsilon;\Lambda \vdash \varepsilon \rightsquigarrow \delta$, univariate cases.

Univariate transforms We use *univariate* for RV transforms between spaces that are not product spaces. The correctness of rules P-LOG, P-EXP, P-LINEAR, and P-SCALE is given by the following lemma.

Lemma 2.5.1. *For absolutely continuous distributions $\mathbb{P}$ and $\mathbb{Q}$ on $\mathbb{R}$ and a function $h : \mathbb{R} \to \mathbb{R}$ such that $\mathbb{Q}(Y) = \mathbb{P}(h^{-1}[Y])$, if h is strictly increasing, differentiable and invertible, then the function*

$$g(y) = f(h^{-1}(y)) \cdot \frac{d}{dy} h^{-1}(y).$$

is a PDF *of $\mathbb{Q}$, where f is the derivative of the* CDF *F of $\mathbb{P}$.*

Proof. The derivative of a CDF is a PDF. The CDF G of $\mathbb{Q}$ is

$$\begin{aligned} G(y) = \mathbb{Q}(\ (-\infty, y]\ )\ &=\ \mathbb{P}(h^{-1}[\ (-\infty, y]\ ]) \\ = \mathbb{P}(\ (-\infty, h^{-1}(y)]\ )\ &=\ F(h^{-1}(y)), \end{aligned}$$

where we have used the fact that the h-preimage of $(-\infty, y]$ is $(-\infty, h^{-1}(y)]$ because h is strictly increasing and invertible. The claim follows from the fact that g is the derivative of G. □

The lemma is easily modified for P-NEG and also P-INV; an "extra" minus sign appears because they consist of strictly *decreasing* components. It is possible to define a version

of P-SCALE for negative literals, as well as integer versions of P-NEG, P-LINEAR, and P-SCALE. With these rules (and P-VAR, discussed below) we can already compute some continuous PDFs. Consider the standard exponential from Section 2.3.2; we derive its PDF with $\emptyset;\emptyset \vdash \mathsf{std\_exponential} \curvearrowright \delta$, which builds the contexts $\Lambda := u$ and $\Upsilon := u : \mathsf{R} \sim \mathsf{random}$ and invokes the chain

$$\begin{aligned}
&\Upsilon;\Lambda \vdash -\mathsf{log}\ u \rightsquigarrow \delta && \delta'' := \lambda x'' \,.\, \langle 0 < x'' < 1 \rangle \\
&\Upsilon;\Lambda \vdash \mathsf{log}\ u \rightsquigarrow \delta' && \delta' := \lambda x' \,.\, \langle 0 < \mathsf{exp}\ x' < 1 \rangle * \mathsf{exp}\ x' \\
&\Upsilon;\Lambda \vdash u \rightsquigarrow \delta'' && \delta := \lambda x \,.\, \langle 0 < \mathsf{exp}\ (-x) < 1 \rangle * \mathsf{exp}\ (-x).
\end{aligned}$$

We β-reduce for clarity. The chain ends with P-VAR, which gives the PDF of Uniform(0,1) for δ''; then, P-LOG and P-NEG produce δ' and δ. The latter is equivalent to $\lambda x \,.\, \langle 0 < x \rangle * \mathsf{exp}\ (-x)$, which is easily seen to be the PDF of the standard exponential. Likewise, the PDF of $\mathsf{uniform}\ \varepsilon_1\ \varepsilon_2$ is correctly calculated to be

$$\delta := \lambda x \,.\, \langle 0 < (x - \varepsilon_1)/(\varepsilon_2 - \varepsilon_1) < 1 \rangle * (1/(\varepsilon_2 - \varepsilon_1)),$$

which is equivalent to $\lambda x \,.\, \langle \varepsilon_1 < x < \varepsilon_2 \rangle * (1/(\varepsilon_2 - \varepsilon_1))$. We do not provide rules for sin, cos, and tan because we are unaware of any simple closed-form expression for the corresponding PDFs.

Multivariate transforms We use *multivariate* for RV transforms to or from a product space. The presence of multiple dimensions introduces the issue of dependence between the inputs or between the outputs of the transform, making it difficult to provide rules that work in the general case. As a result, some of the following rules introduce specific independence requirements.

Rule P-LIT states that the PMF of a point mass distribution on l is simply the indicator function on $\{l\}$. The transforms corresponding to the rules in this section tend to be less obvious; the transform in question for P-LIT is the constant function on l, whose argument may be a tuple. Rule P-BOOL calculates the PMF of a Boolean random variable, which is a simple expression of the probability that the random variable is true. We thus invoke the probability compiler in the current context to compute this probability δ. This rule covers

$$\frac{\emptyset \vdash l : \tau \qquad \tau \text{ countable}}{\Upsilon; \Lambda \vdash l \rightsquigarrow \lambda x : \tau \,.\, \langle x = l \rangle} \text{ P-LIT}$$

$$\frac{\Upsilon \vdash \varepsilon : \mathsf{bool} \qquad \Upsilon; \Lambda \vdash \varepsilon \;\$\; \lambda x : \mathsf{bool} \,.\, \langle x \rangle \mapsto \delta}{\Upsilon; \Lambda \vdash \varepsilon \rightsquigarrow \lambda x : \mathsf{bool} \,.\, \mathsf{if}\ x\ \mathsf{then}\ \delta\ \mathsf{else}\ 1 - \delta} \text{ P-BOOL}$$

$$\frac{\{\Upsilon; \Lambda \vdash \varepsilon_1 \perp \varepsilon_i\}_{i=2,3} \quad \{\Upsilon; \Lambda \vdash \varepsilon_i \rightsquigarrow \delta_i\}_{i=1,2,3}}{\begin{array}{l}\Upsilon; \Lambda \vdash \mathsf{if}\ \varepsilon_1\ \mathsf{then}\ \varepsilon_2\ \mathsf{else}\ \varepsilon_3 \rightsquigarrow \\ \qquad \lambda x \,.\, \delta_1\ \mathsf{true} * \delta_2\ x + \delta_1\ \mathsf{false} * \delta_3\ x\end{array}} \text{ P-IF}$$

$$\frac{\Lambda = \{x\} \sqcup \{y_1, ..., y_m\} \qquad \mathcal{J}(\Upsilon; \Lambda) \mapsto \delta}{\Upsilon; \Lambda \vdash x \rightsquigarrow \lambda x \,.\, \int \lambda(y_1, ..., y_m) \,.\, \delta} \text{ P-VAR}$$

$$\frac{\Lambda = \{x_1, ..., x_n\} \sqcup \{y_1, ..., y_m\} \qquad \mathcal{J}(\Upsilon; \Lambda) \mapsto \delta}{\Upsilon; \Lambda \vdash (x_1, ..., x_n) \rightsquigarrow \lambda(x_1, ..., x_n) \,.\, \int \lambda(y_1, ..., y_m) \,.\, \delta} \text{ P-VARS}$$

$$\frac{\Upsilon; \Lambda \vdash \varepsilon \rightsquigarrow \delta}{\Upsilon; \Lambda \vdash \mathsf{fst}\ \varepsilon \rightsquigarrow \lambda x \,.\, \int \lambda y \,.\, \delta\ (x, y)} \text{ P-FST}$$

$$\frac{\Upsilon; \Lambda \vdash \varepsilon_1 \perp \varepsilon_2 \qquad \{\Upsilon; \Lambda \vdash \varepsilon_i \rightsquigarrow \delta_i\}_{i=1,2}}{\Upsilon; \Lambda \vdash (\varepsilon_1, \varepsilon_2) \rightsquigarrow \lambda(x_1, x_2) \,.\, \ \delta_1\ x_1 * \delta_2\ x_2} \text{ P-PAIR}$$

$$\frac{\Upsilon; \Lambda \vdash \varepsilon_1 \perp \varepsilon_2 \qquad \{\Upsilon; \Lambda \vdash \varepsilon_i \rightsquigarrow \delta_i\}_{i=1,2}}{\Upsilon; \Lambda \vdash \varepsilon_1 + \varepsilon_2 \rightsquigarrow \lambda x : \mathsf{R} \,.\, \int \lambda t : \mathsf{R} \,.\, \ \delta_1\ t * \delta_2\ (x - t)} \text{ P-PLUS}$$

Figure 10: The transform-to-PDF converter, multivariate cases.

$$\frac{}{\mathcal{J}(\Upsilon; \emptyset) \mapsto 1} \text{ J-NIL} \qquad \frac{\Upsilon; \emptyset \vdash e \curvearrowright \delta \qquad \mathcal{J}(\Upsilon; \Lambda) \mapsto \delta'}{\mathcal{J}(\Upsilon, x : \tau \sim e; \Lambda, x) \mapsto \delta\ x * \delta'} \text{ J-CONS}$$

Figure 11: The joint PDF body constructor, $\mathcal{J}(\Upsilon; \Lambda) \mapsto \delta$.

the cases for $<$ and $=$. The ability to represent the PMF of a Boolean random variable allows us to encode arbitrary probability queries. Rule P-IF computes the PDF of a mixture, which is a weighted combination of the component PDFs, where the mixing probability is the probability the if-condition is true. For this to be valid, the if-condition must be independent of its branches, as required. For instance, the PDF of

```
var x ~ random in
    var y ~ uniform 2 3 in return (if x < 1/2 then x else y).
```

is *not* equivalent to $\lambda x \,.\, (1/2) * \langle 0 < x < 1 \rangle + (1/2) * \langle 2 < x < 3 \rangle$, as would be calculated without the restriction (there should be no probability mass on $[1/2, 1]$).

Rule P-VAR is a special case of P-VARS. The transform corresponding to P-VARS is a function that returns a permutation of a subset of components of its tuple argument. We

assume $x_1, ..., x_n$ and $y_1, ..., y_m$ are distinct, and we use $\sqcup$ to denote disjoint union. The resulting PDF is a marginal PDF. The *marginal* PDF of a joint PDF f on $A \times B$ is given by $g(x) = \int \lambda y \,.\, f(x, y)\ d\overline{\mu}_B$; g is a PDF on A whose density at x is given by adding up the contribution of the joint PDF along the other dimension, B. The corresponding process is one which generates tuples but then discards the second component, returning the first. We generalize to higher dimensions by integrating out random variables not appearing in the result tuple. When this set is empty ($m = 0$), the integral reduces to δ. The resulting PDF may be computationally inefficient due to a large number of nested integrals. More efficient schemes that take advantage of the graphical structure of the probabilistic model, such as *variable elimination*, are possible [67]. The judgment $\mathcal{J}(\Upsilon; \Lambda) \mapsto \delta$ constructs the body of the joint PDF of the active random variables (Figure 11). Rule J-CONS first computes the PDF of e, parametric in all of the preceding random variables (thus, invoking the distribution-to-PDF converter with no active random variables). It then constructs the product with the PDFs of the remaining active variables; the product of these parametric PDFs is the joint PDF. The terms δ and δ' in J-CONS have type $\tau \to \mathsf{R}$ and R, respectively. The judgment returns an open term and relies on the fact that the free variables will be bound appropriately by the invoking judgment. Rule P-FST is analogous to P-VARS; we ask for a PDF and compute the marginal PDF of the first component. We define an analogous rule for snd. Rules P-PAIR and P-PLUS state the well known results that the joint PDF and the PDF of the sum of independent random variables is the product of and convolution of their individual PDFs, respectively.

On the face of it, these rules handle mixture models and joint models, but where they really shine is on general hierarchical models. For example, the PDF of

$$\mathsf{hier} := \mathsf{var}\ x \sim \mathsf{random}\ \mathsf{in}\ \mathsf{var}\ y \sim \mathsf{uniform}\ 0\ x\ \mathsf{in}\ \mathsf{return}\ y$$

is not immediately obvious. The process is generated by sampling a value x uniformly from (0,1), and then sampling uniformly from $(0, x)$, discarding x. We calculate the PDF with $\emptyset; \emptyset \vdash \mathsf{hier} \curvearrowright \delta$, which builds $\Upsilon := y : \mathsf{R} \sim \mathsf{uniform}\ 0\ x, x : \mathsf{R} \sim \mathsf{random}$ and $\Lambda := y, x$ for

$\Upsilon; \Lambda \vdash y \rightsquigarrow \delta$. Rule P-VAR then produces

$$\lambda y \,.\, \int \lambda x \,.\, (\langle 0 < (y-0)/(x-0) < 1 \rangle * 1/(x-0)) * \langle 0 < x < 1 \rangle * 1$$

for δ, where we have β-reduced for clarity. The body of the inner λ-abstraction is generated by the joint PDF body constructor; the two non-trivial multiplicands are the parametric PDF of $\mathsf{uniform}\ 0\ x$ and the PDF of random, respectively. With some manipulation we can show δ corresponds to $f(y) = \int_y^1 1/x \; dx = -\log(y)$ for $y \in (0, 1)$ and zero otherwise. The rules do not perform algebraic simplifications, but the benefit of automation can still be felt clearly.

Modularity Some RV transforms are inconvenient to work with, preventing us from calculating certain PDFs. For example, we cannot calculate the PDF of $\mathsf{std\_normal}$ from scratch because its specification uses cos, which we do not handle. However, the design allows us to modularly address cases like this, where we want to specially handle the PDF for a specific distribution. We can add the rule $\Upsilon; \Lambda \vdash \mathsf{std\_normal} \curvearrowright \phi$, where $\phi := \lambda x \,.\, \mathsf{exp}(-x * x/2)/\mathsf{sqrt}(2 * \pi)$ is the PDF of the standard normal. This new rule is used by the joint body constructor whenever $\mathsf{std\_normal}$ appears on the spine of a distribution, enabling the calculation of PDFs for hierarchical models using $\mathsf{std\_normal}$ that were previously not compilable. For example, the PDF of $\mathsf{normal}\ \mu\ \sigma$ can now be calculated as

$$\begin{aligned}
&\Upsilon; \Lambda \vdash \sigma * x + \mu \rightsquigarrow \delta && \delta'' := \lambda x'' \,.\, \phi\ x'' \\
&\Upsilon; \Lambda \vdash \sigma * x \rightsquigarrow \delta' && \delta' := \lambda x' \,.\, \phi\ (x'/\sigma) * (1/\sigma) \\
&\Upsilon; \Lambda \vdash x \rightsquigarrow \delta'' && \delta := \lambda x \,.\, \phi\ ((x - \mu)/\sigma) * (1/\sigma)
\end{aligned}$$

using the rules P-VAR, P-SCALE, and P-LINEAR, where $\Lambda := x$ and $\Upsilon := x : \mathsf{R} \sim \mathsf{std\_normal}$. We can see δ is equivalent to the classic formula for the normal PDF, $f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp(-\frac{1}{2\sigma^2}(x-\mu)^2)$. Likewise, we can now handle distributions like the log-normal and mixture-of-Gaussians. To support an infinite discrete distribution with a known PDF, such as the Poisson distribution, we can add a new primitive to the core calculus ($\mathsf{poisson}\ \varepsilon$) and handle it specially in the distribution-to-PDF converter.

2.6 Empirical evaluation

We implement an enhanced version of the PDF compiler in the probabilistic language Infer.NET Fun [11], which is embedded in F# [63]. The enhancements include rules for handling arrays and records (based on the rules for tuples), `fail` statements, `match` and general `if` expressions, and—for performance reasons—deterministic `let` expressions. The implementation is detailed in [8], which also describes a proof of soundness for the compiler.

We evaluate the compiler on synthetic textbook examples and real examples from scientific applications that use *Markov chain Monte Carlo* (MCMC) for performing Bayesian inference. We wish to validate that the PDF compiler handles these examples and understand how it reduces the developer burden as well as its performance impact.

MCMC methods are commonly used for Bayesian inference and generate samples from the posterior distribution. The idea of MCMC is to construct a Markov chain in the parameter space of the model, whose equilibrium distribution is the posterior distribution over model parameters. Neal [45] gives an excellent review of MCMC methods. We here use Filzbach [52], an adaptive MCMC sampler based on the Metropolis-Hastings algorithm. All that is required for such algorithms is the ability to calculate a function proportional to the posterior density, given a set of parameters. The posterior does not need to be from a mathematically convenient family of distributions. Samples from the posterior can then serve as its representation, or be used to calculate marginal distributions of parameters or other integrals under the posterior distribution.

The posterior density is proportional to the product of the likelihood function of the data and the density function of the prior distribution. Filzbach and other MCMC libraries require users to write these two functions, in addition to the probabilistic generative functions used to generate synthetic data, which are used for model validation. With our PDF compiler, we can instead compile these density functions from the generative code. This relieves domain experts from having to write the density code in the first place, as well as from the error-prone task of manually keeping their model code and their density code in synch. Instead, both the PDF and synthetic data are derived from the same declarative specification of the model.

Implementation. Since Fun is a sublanguage of F#, we implement our models as F# programs and use the quotation mechanism of F# to capture their syntax trees. Running the F# program corresponds to sampling data from the model. To compute the PDF, the compiler takes the syntax tree (of F# type `Expr`) of the model and produces another `Expr` corresponding to a deterministic F# program as output. We then use run-time code generation to compile the generated `Expr` to MSIL bytecode, which is just-in-time compiled to executable machine code when called, just as for statically compiled F# code. Our implementation supports arrays and records, which are both translated using adaptations of the corresponding rules for tuples. For efficiency, the implementation must avoid introducing redundant computations, translating the use of substitution in the formal rules to more efficient `let`-bindings that share the values of expressions that would otherwise be re-computed. As is common practice, our implementation and Filzbach both work with the *logarithm* of the density, which avoids products of densities in favor of sums of log-densities where possible, to avoid numerical underflow.

Metrics. We consider scientific models with existing implementations for MCMC-based inference, written by domain experts. We are interested in how the modelling and inference experience would change,in terms of developer effort and performance impact, when adopting the Fun-based solution.

We assess the reduction in developer burden by measuring the code sizes (in lines-of-code (LOC)) of the original implementations of model and density code, and of the corresponding Fun model. For the synthetic examples, we have written both the model and the density code. The original implementations of the scientific models contain helper code such as I/O code for reading and writing data files in an application-specific format. Our LOC counts do not consider such helper code, but only count the code for generating synthetic data from the model, code for computing the logarithm of the posterior density of the model, and model-related code for setting up and interacting with Filzbach itself. We also compare the running times of the original implementations versus the Fun versions, not including data manipulation before and after running inference.

Table 1: Lines-of-code and running time comparisons of synthetic and scientific models.

Example	**orig**	**loc**	**loc, Fun**		**time**	**time, Fun**	
mixture of Gaussians	F#	32	20	0.63x	1.77	4.78	2.7x
linear regression	F#	27	18	0.67x	0.63	2.08	3.3x
species distribution	C#	173	37	0.21x	79	189	2.4x
net primary productivity	C#	82	39	0.48x	11	23	2.1x
global carbon cycle	C#	1532	402	0.26x	–	764	–

2.6.1 Examples

Synthetic examples. Our synthetic examples are models for two classic problems in statistics and machine learning: the supervised learning task *linear regression*, and the unsupervised learning task *mixture of Gaussians*. The latter can be thought of as a probabilistic version of *k-means clustering*. In linear regression, inference is trying to determine the coefficients of the line. In mixture of Gaussians, inference is trying to determine the unknown mixing bias and the means and variances of the Gaussian components.

Species distribution. The species distribution problem is to give the probability that certain species will be present at a given site, based on climate factors. It is a problem of long-standing interest in ecology and has taken on new relevance in light of the issue of climate change. The particular model that we consider is designed to mitigate *regression dilution* arising from uncertainty in the predictor variables, for example, measurement error in temperature data [38]. Inference tries to determine various features of the species and the environment, such as the optimal temperature preferred by a species, or the true temperature at a site.

Global carbon cycle. The dynamics of the Earth's climate are intertwined with the terrestrial carbon cycle, and better carbon models (modelling how carbon in the air gets converted to biomass) enable better constrained projections about these systems. We consider a fully data-constrained terrestrial carbon model by [61]. It is a composition of various submodels for smaller processes such as *net primary productivity*, the fine root mortality rate or the fraction of trees that are evergreen versus deciduous. Inference tries to determine

the different parameters of these submodels.

Discussion. Table 1 reports the metrics for each example. The LOC numbers show significant reduction in code size, with more significant savings as the size of the model grows. The larger models (where the Fun versions are $\approx 25\%$ of the size of the original) are more indicative of the savings in developer and maintenance effort, since smaller models have a larger fraction of boiler-plate code. We find the running times encouraging: we have made little attempt to optimize the generated code, and preliminary testing indicates that much of the performance slow-down is due to constant factors.

The global carbon cycle model is composed of submodels, each with their own dataset. Unfortunately, it is unclear from the original source code how this composition translates to a run of inference, making it difficult to know what constitutes a fair comparison. Thus, we do not report a running time for the full model. However, we can measure the running time of individual submodels, such as net primary productivity, where the data and control flow are simpler.

2.7 Related work

Our work builds on a long tradition of probabilistic functional languages, most connected to the probability monad in some way. They work by incorporating distributional semantics into a functional language, so that one can express values which represent a *distribution* over possible outcomes. The distribution can either be manifest (available to the programmer) or implicit (existing only in the metatheory). An early incarnation of the latter was given by Kozen in [34], in which he provides the semantics for an imperative language endowed with a random number primitive supplying samples from Uniform(0,1). Values of type A in the object language are given semantics in functions of type $(A \to [0,1]) \to [0,1]$ in the metatheory. These functions represent distributions over A and satisfy the expected laws for measures. Kozen's work is far-reaching and will continue to inspire future languages: it can accommodate continuous and hybrid distributions; it handles unbounded iteration (general recursion), a traditionally thorny issue for probabilistic languages; and it even provides a treatment of distributions on function types. However, PDFs are not addressed at all.

Though not explicitly cast as functional or monadic, Kozen's approach forms the basis for Audebaud and Paulin-Mohring's monadic development for reasoning about randomized algorithms in Coq [3]. Their focus is on verification, and they define the probability monad from first principles (modulo an axiomatization of arithmetic on the [0,1] interval), whereas we provide it axiomatically. We hope to inspire a cross-fertilization of ideas between the efforts as we bring our theory of PDFs into Coq.

While suitable for semantics and verification, Kozen's representation is not ideal for direct use in computing certain operations. For instance, it is unclear how to sample or compute general expectations efficiently given a term of type $(A \to [0,1]) \to [0,1]$. More recent works explore alternate concrete embodiments of the probability monad; Ramsey and Pfeffer discuss some of the possibilities [55]. A popular choice is to represent distributions as weighted lists or trees. This has the drawback that only distributions with finitely many outcomes are expressible (ruling out essentially all commonly used continuous distributions), and PMFs are the only supported form of PDFs. On the other hand, distributions can occur on arbitrary types, expectation and computing the PMF is straightforward, and the approach works well as an embedded domain-specific language (PFP [19], HANSEI [32], probability monads in Haskell [55]). Dedicated languages like IBAL [50] or Church [23] offer more scope for program analysis, which is crucial for escaping the limitations of an embedded approach and mitigates some of the fundamental drawbacks of the representation. Ultimately, however, these languages do not support continuous or hybrid distributions (nor their PDFs) in a general sense. Although, inference for Church also uses MCMC, but works with distributions over the runs of a program instead of over its return value [68].

Sampling functions are a fun alternative representation. They are used by $\lambda_{\bigcirc}$ [48] to support continuous and hybrid distributions in a true sense and also allow distributions on arbitrary types. Distributions are represented by sampling functions that return a sample from the distribution when requested. Sampling and sampling-based routines are the only supported operations, thus PDFs are not accommodated.

Another recent work also rigorously supports continuous and hybrid distributions by providing a measure transformer semantics for a core functional calculus [11]. The work

does not provide PDFs but is novel for its ability to support conditional probability in the presence of zero probability events in continuous spaces, a feature necessary in many machine learning applications. Their formalization is similar to ours, as both are based in standard measure theory. They have independently recognized the importance of analyzing distributions by their transformations, doing so in the context of conditional probability, whereas we have developed the idea for PDFs. This hints that reasoning via transforms may be a technique that is more broadly applicable to other program analyses for probabilistic languages.

The Hierarchical Bayes Compiler (HBC) is a toolkit for implementing hierarchical Bayesian models [15]. Its specification language represents a different point in the design space. Essentially, it removes return while adding a set of standard distributions (with PDFs) to the core calculus. This guarantees that all constructible models are AC. Many powerful models used in machine learning are expressible in HBC. However, something as basic as adding two random variables is not. Furthermore, if a distribution outside of the provided set is required, it must be added to the core. This is the fundamental tension surrounding return: with it, the core is minimal, expressivity is high, and PDFs are non-trivial; without it, PDFs are easily supported, but the core becomes large, and expressivity is crippled. HBC is not formally defined.

An entirely different tradition incorporates probabilistic semantics into logic programming languages (Markov Logic [56], BLOG [41], BLP [31], PRISM [57]). These languages are well suited for probabilistic knowledge engineering and statistical relational learning. In Markov Logic, for instance, programmers associate higher weights with logical clauses that are more strongly believed to hold. The semantics of a set of clauses is given by *undirected graphical models*, with the weights determining the potential functions (*e.g.* by Boltzmann weighting). Certain continuous distributions can be supported by manipulating the potential function calculation. Supporting PDFs in this context should not be problematic; the potential functions (essentially, unnormalized PDFs) always exist, by design. However, like HBC, it appears these languages are not quite as expressive as is possible in a probabilistic functional language.

The AutoBayes system [24] shares a key feature with our language in that PDFs are manifest in the object language. AutoBayes automates the derivation of maximum likelihood and Bayesian estimators for a significant class of statistical models, with a focus on code generation, and can express continuous distributions and PDFs. However, despite their focus on correctness-by-construction, the language is not formally defined. Furthermore, it is unclear how general the language actually is, *i.e.* how "custom" the models can be. Our work could serve as a formal basis for their system.

2.8 Conclusion

We have presented a formal language capable of expressing discrete, continuous and hybrid distributions and their PDFs. Our novel contributions include a type system for absolutely continuous distributions and a modular PDF calculation procedure. The type system uses the new ideas of RV transforms and non-nullifying functions. There are several interesting avenues for future work. The first is to address PDFs in the context of conditional probability, perhaps by incorporating our formalization of PDFs with the ideas presented in [11]. Secondly, to provide a complete account of continuous probability, one must support expectation. Generically supporting expectation requires a treatment of integrability or summability; reasoning via the RV transform may be a productive route. Finally, combining this work with a formal language for optimization such as [2] would create a true formal language for *statistics*, which would be able to express statistical problems in the object language itself. Current languages express *probability*; any notion of statistics is outside the language.

CHAPTER III

A SYNTACTIC THEORY OF OPTIMIZATION

Mathematical programs (MPs) are a class of constrained optimization problems that include linear, mixed-integer, and disjunctive programs as well as other forms. Strategies for solving MPs rely heavily on various transformations between these subclasses. There is a great need to automate these transformations because they are algebraically cumbersome and require expertise in specialized topics. However, most are currently not automated because MP theory does not practice treating programs as syntactic objects. In this chapter, we review Tyles [1, 2], the first syntactic definition of MP. Building on this, we provide a novel formalization of the big-M method, in the style of the convex-hull formalization given in Tyles. Both of these are widely-used transformations for reformulating disjunctive constraints. Finally, we implement both transformations and compare with state-of-the art solutions. We have implemented our object language as an embedded domain-specific language in OCaml and use it to experiment with the different transformations provided.

3.1 Introduction

The equations governing engineering systems rarely dictate a unique solution. Usually, a designer needs to find the optimal solution amongst a space of feasible ones. Such constrained optimization problems are often expressed as mathematical programs (MPs), which consist of a numerical objective that is to be maximized (or minimized) subject to some constraints. Solving MPs efficiently is an important problem across science and engineering. The nature of the constraints allowed is a key issue affecting both the kinds of systems that can be represented and the efficiency of algorithms. An MP is more specifically called a linear program (LP) when the constraints are linear algebraic equations and inequalities on the reals. A mixed-integer linear program (MILP) additionally allows restricting variables to be integer valued, which allows expressing problems not possible in LP. We discuss a superset of these that also allows Boolean expressions and most importantly disjunctive constraints.

Throughout this work, the term *disjunctive constraint* refers to a disjunction over (in)equations involving reals, and is unrelated to Boolean disjunction which is a statement purely over Boolean variables. Both are an important modeling tool, and transforming disjunctive constraints is especially challenging. Consider the designer of a chemical plant who must decide between one of two reactors to purchase. The reactors operate in different regimes: Reactor 1 operates under greater temperature and pressure than Reactor 2. Higher temperature and pressure lead to an increased reaction rate and thus increased monthly revenue; however, they also come with higher operating costs. The operating regimes are pictured in Figure 12a and are described by

$$R^1 = \begin{bmatrix} x_1 \geq 1 \\ x_2 \geq 1 \\ x_1 + x_2 \leq 5 \end{bmatrix} \quad R^2 = \begin{bmatrix} 5 \leq x_1 \leq 8 \\ 4 \leq x_2 \leq 7 \end{bmatrix} \tag{4}$$

where x_1 and x_2 correspond to temperature and pressure. Our goal is to maximize profit. An intuitive way to express this problem is as a disjunctive program:

$$\max \left\{ \text{profit}(x_1, x_2) \mid R^1 \vee R^2;\ x_1, x_2 \in \mathbb{R} \right\} \tag{5}$$

This represents finding a point which satisfies the constraint R^1 *or* the constraint R^2 and for which profit is maximized.

Unfortunately, MP solvers do not directly accept disjunctive programs as input. A naive approach of addressing this problem is to perform two separate optimizations (one for each region individually) and take the maximum, but this does not scale well as the constraints take more complex forms (e.g. nested disjunctions, additional logical conditions between constraints). The currently best-known strategies reformulate the program into an equivalent MILP, for which there are good solvers.

One such efficient reformulation technique is Balas' convex-hull method [5]. Applying

this method to our problem yields the constraint

$$\begin{bmatrix} x_1^A \geq y^A \\ x_2^A \geq y^A \\ x_1^A + x_2^A \leq 5y^A \end{bmatrix} \wedge \begin{bmatrix} 5y^B \leq x_1^B \leq 8y^B \\ 4y^B \leq x_2^B \leq 7y^B \end{bmatrix} \tag{6a}$$

$$x_1 = x_1^A + x_1^B \tag{6b}$$

$$x_2 = x_2^A + x_2^B \tag{6c}$$

$$1 = y^A + y^B \tag{6d}$$

which interestingly has no disjunctions. We will see later how this constraint is equivalent to $R^1 \vee R^2$ but for now focus on the mechanization challenges. Note that several new variables had to be introduced, the original constraints had to be modified, and some new equations are added. Consider the general case of a disjunction with n variables, k disjuncts, and an average of m inequalities per disjunct. The convex-hull method requires generating $kn + k$ new variables, manipulating km inequalities, and creating $n+1$ new equations. Remarkably, neither the theoretical definitions of MP nor MP software support locally scoped variable declarations. The numerous variable names generated must thus be unique across the entire program.

In practice, k and m are magnified further for two reasons. Firstly, Balas' theory requires each disjunct to be bounded, which often is attained by adding a lower and upper bound for every variable in each disjunct. This increases the number of inequalities per disjunct from m to $m+2n$. Secondly, the method applies only to disjunctions in disjunctive normal form. Nested disjunctions can be first converted to DNF, compounding the number of disjuncts, or the convex-hull method can be applied iteratively starting from the inner-most disjunction, compounding the number of inequalities within outer disjuncts at each step.

The reformulation is error-prone not just because of the tedious algebra, but also because the resulting equations are non-intuitive. Even on small problems, it is challenging to recognize how the output represents the original constraint. Finally, one must of course be familiar with the reformulation methods to apply them. Automation is clearly called for.

The reformulations we present have been widely used by experts for many years. However, there has been limited to no support for them in MP software tools. This is because current MP theory focuses on the study of the numerical behavior of algorithms and does not treat programs as syntactic objects. MPs are defined in a canonical matrix form, which does not support basic operations required for automating transformations such as variable introduction and compositional construction of programs. Tyles demonstrates that the formal methods of language design capably address long standing needs in the mathematical programming community. In this chapter, we present more evidence of the benefits of the formal approach. Specifically, our contributions in contrast with Tyles are:

- We formalize the big-M method for reformulating disjunctive constraints. This is done in the style of the convex-hull formalization given in Tyles. The fact that we are so easily able to implement the big-M method in Tyles—despite the fact that it has resisted a successful implementation for so long—showcases the power of the formal approach to language design.

- The original software in Tyles implements the concept of treating MPs as syntactic objects, but does not connect to numerical solvers. We provide such an implementation, which comes in two parts: an embedded domain-specific language (EDSL) in OCaml for succinct construction of MPs, and implementations of the big-M and convex-hull reformulations. Our software outputs programs in the popular AMPL language and the industry standard MPS format. This allows us to pass the programs generated by our software to existing solvers and study their behavior. We find that our software generates efficient programs.

3.2 Background

3.2.1 Mathematical programming

The standard definition of a linear program is

$$\max \{cx \mid Ax \leq b, x \in \mathbb{R}^n\} \tag{7}$$

where c is a $1 \times n$ dimensional coefficient vector, x is an $n \times 1$ vector of real valued variables, A is an $m \times n$ coefficient matrix, and b is an $m \times 1$ vector of constants. Thus, cx is a scalar, and the matrix inequality $Ax \leq b$ represents m individual inequalities. The inequalities represent a polyhedron, such as either region R^1 or R^2 in Figure 12a, and is called the feasible space of the LP.

Representing discrete choices requires a more expressive language than LP. We need a language that allows expressing not just R^1 or R^2 separately but their union $R^1 \cup R^2$. There are two rather distinct methods for accomplishing this. The first is to enrich LP with a discrete type, such as is done with mixed-integer linear programming (MILP). In MILP, variables may be integer or real valued. The standard definition [46] is

$$\max\{cx + hy \mid Ax + Gy \leq b, x \in \mathbb{R}^n, y \in \mathbb{Z}^p\} \tag{8}$$

where x and y represent vectors of real and integer variables, respectively.

However, integers are often not an intuitive model of discrete choice, and become prohibitively difficult for larger problems. Alternatively, LP can be enriched with disjunctive constraints, which lead to more compact and comprehensible models [5, 54]. The canonical matrix form of a disjunctive constraint is

$$\left[A^1 x \leq b^1\right] \vee \left[A^2 x \leq b^2\right] \tag{9}$$

We still do not have Boolean expressions, nor disjunctive constraints not in DNF, nor methods for introducing locally scoped variables, nor an obvious way to insert new constraints or extract specific ones to manipulate. In short, these definitions do not provide an abstract syntax that can be operated on formally. In this section, we define such a syntax and include brief discussion of the fairly straightforward type system and semantics.

3.2.2 Review of Tyles: a language for mathematical programming

The language Tyles is a formal language for expressing mathematical programs. It consists of types τ, expressions e, constraints c (called *propositions* in logic), and programs p. The

syntax is

$$\tau ::= \texttt{real} \mid \texttt{bool} \tag{10a}$$

$$e ::= x \mid r \mid \texttt{true} \mid \texttt{false} \mid \texttt{not}\, e \mid e_1\, \texttt{or}\, e_2 \mid e_1\, \texttt{and}\, e_2 \mid -e \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \tag{10b}$$

$$c ::= \texttt{T} \mid \texttt{F} \mid \texttt{isTrue}\, e \mid e_1 = e_2 \mid e_1 \leq e_2 \mid c_1 \vee c_2 \mid c_1 \wedge c_2 \mid \exists x{:}\tau \,\texttt{.}\, c \tag{10c}$$

$$p ::= \max_{x_1:\tau_1,\ldots,x_m:\tau_m} \{e \mid c\} \tag{10d}$$

A mathematical program p consists of an objective e that must be maximized subject to a constraint c. Minimizing is equivalent to maximizing $-e$. This definition is similar to (8) but the objective and constraint are not in a matrix form.

Disregarding `isTrue` e for the moment, constraints are essentially a conjunction or disjunction over (in)equations on the reals. Disjunction $c_1 \vee c_2$ is the key novelty. Conjunction alone provides a language for expressing what is normally referred to as a system of linear equations in linear algebra.

Although it is not common in the MP literature, variables must be explicitly introduced with an existential quantifier. This clarifies the semantics and provides the practical benefit of locally scoped variables. Universal quantifiers would extend the language to include semi-infinite programs, an interesting but less developed class of problems. Variables introduced at the program level behave as existentially quantified; the only distinction being that they can also be used in the objective.

In addition, we have Boolean propositions in the form `isTrue` e, where e must be an expression of type `bool`. There is a distinction between Boolean truth versus truth of numeric propositions (`true` and `false` versus `T` and `F`). This type distinction, embodied as a syntactic distinction in the definition, is essential since the algorithms for solving these classes of propositions are entirely different. The convex-hull and big-M methods are useful only for the disjunctive constraint $c_1 \vee c_2$ and should not be applied to the Boolean expression $e_1\, \texttt{or}\, e_2$. Additionally, Boolean expressions can be negated, but there is no negation at the constraint level because MPs do not allow strict inequalities.

Expressions are categorized into the types `real` and `bool`; integers will be discussed shortly. They include variables, rational constants r, Boolean constants, and the usual numeric and Boolean operators. We wish only to support linear terms, and so the restriction on $e_1 * e_2$ is that e_1 has no free variables. Nonlinear programs are certainly important, but the transformations we are focusing on apply only to linear constraints.

Agarwal's thesis [1] details the type system and semantics for an extended version of this language that includes indexing capabilities, but one point is worth clarifying here. Mathematical programs involve real numbers, which raises the issue of computing over them. This is a fundamental challenge being pursued by others in various contexts [51, 44]. It does not however affect the transformations we provide because they are purely syntactic manipulations, and all real expressions are carried through unaltered. We were careful to include only *rational* constants instead of reals in the syntax, but this is due to an unrelated issue. It is a specification of MILPs that constants be rational, else an optimum may not exist [46]. Despite the MP community's classical treatment of reals, it is interesting to note that their desired interpretation of disjunction and existential quantification is certainly constructive. It is expected that any MP solver explain how the constraints are satisfied by providing witnesses for all variables and information on which disjoint region the optimal was found in.

The convex-hull method is applicable only when all disjuncts represent a bounded region, and the big-M method requires computing lower and upper bounds on arbitrary expressions. Thus we need a treatment of bounds, and also we have yet to support integers. These needs are satisfied by introducing refined types

$$\begin{aligned} \rho ::= {} & [r_L, r_U] \mid [r_L, \infty) \mid (-\infty, r_U] \mid \texttt{real} \\ & \mid \langle r_L, r_U \rangle \mid \langle r_L, \infty) \mid (-\infty, r_U \rangle \mid \texttt{int} \\ & \mid \{\texttt{true}\} \mid \{\texttt{false}\} \mid \texttt{bool} \end{aligned} \tag{11}$$

which can be thought of as subsets of types τ. Square brackets denote real intervals, and angle brackets integer intervals. Classically, integers are a subset of the reals.

Bounds can be provided for variables at the time they are introduced by a slight modification of the syntax; type declarations are replaced with refined type declarations. These occur in existential quantifiers and at the program level. Instead of $\exists x : \tau \centerdot c$ we allow $\exists x : \rho \centerdot c$, and instead of $\max_{x_1:\tau_1,\ldots,x_m:\tau_m} \{e \mid c\}$ we allow $\max_{x_1:\rho_1,\ldots,x_m:\rho_m} \{e \mid c\}$.

We keep track of variable bounds with a refined type context

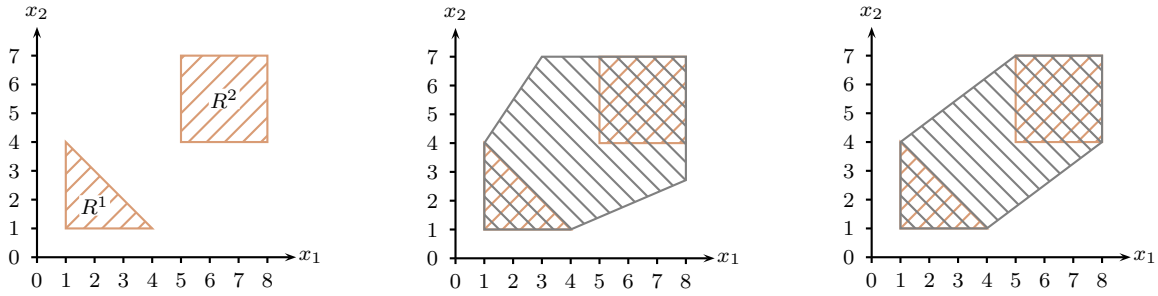
$$\Upsilon ::= \bullet \mid \Upsilon, x : \rho \tag{12}$$

which is a list of variables associated with their bounds. This is more informative than the usual context used in typing judgments. It provides not just variables' types but also retains knowledge of restrictions on the variables' values.

Free variables and substitution are defined in the usual way. Let $FV(e)$ and $FV(c)$ refer respectively to the free variables of an expression and constraint. For example, $FV(\exists x : \rho \centerdot x + (1 - y)) = \{y\}$ because x is bound within the body of an existential constraint. Let $\{e/x\}\, e'$ denote the substitution of e for x in e', handling variable capture as needed. Similarly, $\{e/x\}\, c$ substitutes e for x in constraint c. Programs p are not allowed to have free variables because their definition is not inductive.

3.3 Transforming syntactic constructs

The class of programs covered by p include disjunctive constraints and Booleans, but the best solvers accommodate only mixed-integer linear programming (MILP) constraints which do not allow either of these forms. We pursue the standard strategy of transforming the richer constraint forms to lower-level MILP constraints, with the important distinction that our definitions lead to a software implementation. Transformations for handling Boolean propositions and for using the convex-hull method on disjunctive constraints are described in [1, 2], so we start by immediately presenting the big-M formalization. The target language for our transformations is a subset of the source language, but we also discuss a reformulation to indicator constraints, a new constraint form supported by CPLEX.



(a) Feasible region. (b) Big-M relaxation. (c) convex-hull relaxation.

Figure 12: A disjunctive region and two reformulations.

3.3.1 The big-M transformation

We now turn our attention to transformations for disjunctive constraints $c_1 \vee c_2$. The methods make no use of standard logical laws, such as DeMorgan's (recall constraints cannot be negated), and it is perhaps surprising that it is even possible to eliminate the disjunction entirely. However, we know it is possible, under a mild condition, due to Balas' convex-hull method, the big-M method, and other techniques [5, 4, 54, 29].

The general idea is that the dichotomy expressed by disjunction is embodied instead in the discrete nature of integer variables. An integer binary variable $y_i \in \{0, 1\}$ is associated with each i^{th} disjunct of a disjunction, and the disjunction is replaced by conjunction. Just one y_i is required to be 1 and only the constraints of the corresponding disjunct are enforced. Disjuncts $j \neq i$ get reduced to tautologies.

There are several ways to apply this general approach. A naive method is to multiply the inequalities in each disjunct by its corresponding binary variable. For example, $R^1 \vee R^2$ from our introductory example would become

$$\begin{bmatrix} x_1 y^A \geq y^A \\ x_2 y^A \geq y^A \\ (x_1 + x_2) y^A \leq 5 y^A \end{bmatrix} \wedge \begin{bmatrix} 5y^B \leq x_1 y^B \leq 8y^B \\ 4y^B \leq x_2 y^B \leq 7y^B \end{bmatrix} \tag{13a}$$

$$1 = y^A + y^B \tag{13b}$$

where we have introduced the binary variables y^A and y^B. Unfortunately, this simple transformation is also a poor one. Multiplying two variables creates nonlinear and possibly

nonconvex programs, which are significantly more complex to solve. Both the big-M and convex-hull methods produce linear constraints.

The big-M method states that (9) can be reformulated into the equivalent mixed-integer linear constraints

$$A^1 x - b^1 \leq M^1(1 - y_1) \tag{14a}$$

$$A^2 x - b^2 \leq M^2(1 - y_2) \tag{14b}$$

$$y_1 + y_2 = 1 \tag{14c}$$

where $y_i \in \{0, 1\}$ and M^i are the so called big-M parameters. These are known upper bounds on $A^i x - b^i$. Consider $y_1 = 1$ and $y_2 = 0$. The second inequality reduces to $A^2 x - b^2 \leq M^2$, which is trivially satisfied because, by definition, M^2 is the maximum value the left-hand side could take. Effectively, the second disjunct is disregarded. The first inequality reduces to $A^1 x - b^1 \leq 0$, which is the original first disjunct. Conversely, only the second disjunct is enforced if $y_1 = 0$ and $y_2 = 1$. So these MILP constraints are seen to be equivalent to the original disjunctive constraint.

The computational efficiency of this method is crucially dependent on the choice of the big-M parameters, of which there are quite a few since M^1 and M^2 are vectors. Casual users usually set them to some arbitrarily large value to avoid the effort of computing them. Even experts often resort to this because it preserves model modularity. A tight M implies certain bounds on the left-hand-side variables. Any changes to the bounds specified when the variable was introduced would require one to search through their entire program to verify that all the M's are still valid. A liberally large value avoids this. In contrast, our automation solution preserves modeling simplicity while providing computational efficiency. We use interval arithmetic to compute tight big-M parameters automatically.

Our definition of the big-M method requires two auxiliary judgments to be first introduced. First, we need an operation for computing big-M parameters. Let $\Upsilon \vdash e \rightleftharpoons [r_L, r_U]$ be the judgment that computes lower and upper bounds r_L and r_U for the expression e in the refined context Υ, where r_L and r_U may take on the values of $-\infty$ and ∞. Its definition uses interval arithmetic over unary negation and the binary operators $+$, $-$, and $*$

by propagating derived bounds from subterms to enclosing terms. For example, under the context $x : [-1, 2], y : [0, 100]$, the expression $-5 * x + y$ generates the interval $[-10, 105]$.

Second, we define an operation to convert an inequality to its big-M form. Let $\Upsilon \vdash e \otimes c \rightharpoonup c'$ be the judgment that rewrites constraint c to its big-M form c', where the e will supply the necessary $1 - y$ term. Its definition is

$$\frac{\Upsilon \vdash e_1 - e_2 \rightleftharpoons [r_L, r_U]}{\Upsilon \vdash e \otimes e_1 \leq e_2 \rightharpoonup e_1 \leq e_2 + e * r_U} \tag{15a}$$

$$\frac{\left\{\Upsilon \vdash e \otimes c_j \rightharpoonup c'_j\right\}_{j \in \{A,B\}}}{\Upsilon \vdash e \otimes c_A \wedge c_B \rightharpoonup c'_A \wedge c'_B} \tag{15b}$$

$$\frac{\Upsilon, x : \rho \vdash e \otimes c \rightharpoonup c'}{\Upsilon \vdash e \otimes \exists x : \rho \centerdot c \rightharpoonup \exists x : \rho \centerdot c'} \tag{15c}$$

The first rule is the interesting one. It converts the inequality $e_1 \leq e_2$ by computing bounds for $e_1 - e_2$, where the upper bound is the desired big-M parameter. The lower bound is not needed. This upper bound multiplied by e, which will be of the form $1 - y$, is then added to the appropriate side of the inequality. Conjunctive constraints and existential constraints recurse into their subterms, where in the latter case we add the introduced variable to the context. Other forms need not be defined because they are compiled away prior to applying the big-M method.

A finite upper bound on $e_1 - e_2$ must exist. Our software assures this and prints an informative message when a finite bound cannot be computed.

Finally, we define the main big-M compiler. Let $\Upsilon \vdash c \overset{\text{BIGM}}{\longmapsto} c'$ be a judgment converting a disjunctive constraint c to an MILP constraint c' via the big-M method:

$$\frac{\begin{array}{l}\left\{\Upsilon \vdash c_j \overset{\text{PROP}}{\longmapsto} c'_j\right\}_{j \in \{A,B\}} \\ \Upsilon \overset{\text{CTXT}}{\longmapsto} \Upsilon' \\ \left\{\Upsilon' \vdash (1 - y_j) \otimes c'_j \rightharpoonup c''_j\right\}_{j \in \{A,B\}}\end{array}}{\Upsilon \vdash c_A \vee c_B \overset{\text{BIGM}}{\longmapsto} \left(\begin{array}{l}\exists y_A : \langle 0, 1\rangle \centerdot \exists y_B : \langle 0, 1\rangle \centerdot \\ \quad (y_A + y_B = 1) \wedge (c''_A \wedge c''_B)\end{array}\right)} \tag{16}$$

First, the disjuncts themselves are compiled using the overall constraint compiler $\overset{\text{PROP}}{\longmapsto}$, defined subsequently. This converts the disjuncts to MILP form. Then, in the converted

context, for each disjunct c_j we introduce a corresponding binary variable and rewrite c_j to a big-M form. Finally, the overall result is constructed with appropriate introduction of the y's, the equation forcing the sum of y's to be 1, and with the original disjunction $c_A \vee c_B$ replaced with the conjunction $c''_A \wedge c''_B$.

3.3.2 The indicator constraint transformation

Recently, the CPLEX system has been extended to natively handle a new constraint form known as an *indicator constraint*, which is now supported by languages that interface to it such as AMPL and GAMS. Indicator constraints can be seen as a restricted alternative to disjunctive constraints. They are of the form

$$y = k \Rightarrow e_1 \,\mathtt{op}\, e_2$$

where y is a binary variable, $k \in \{0, 1\}$, and $\mathtt{op} \in \{\leq, =, \geq\}$ i.e. the head of the implication is simply a binary condition and the body is a single numerical relation. The idea is that instead of writing a disjunctive constraint, one can write two indicator constraints, where the heads are mutually exclusive and the bodies represent the disjuncts.

Our constraint from (5) can be represented with indicator constraints as follows:

$$\begin{bmatrix} y^A = 1 \Rightarrow x_1 \geq 1 \\ y^A = 1 \Rightarrow x_2 \geq 1 \\ y^A = 1 \Rightarrow x_1 + x_2 \leq 5 \end{bmatrix} \wedge \begin{bmatrix} y^B = 1 \Rightarrow 5 \leq x_1 \\ y^B = 1 \Rightarrow x_1 \leq 8 \\ y^B = 1 \Rightarrow 4 \leq x_2 \\ y^B = 1 \Rightarrow x_2 \leq 7 \end{bmatrix} \tag{17}$$

$$y^A + y^B = 1 \tag{18}$$

For this toy example, the formulation is reasonably intuitive also, although we feel the disjunctive version is still more natural. However, indicator constraints would be difficult to use on more involved problems with nested disjunctive conditions since they can only occur at the "top level" of a program; they cannot occur even in the body of another indicator constraint. Essentially, one would have to convert disjunctions to DNF before they could be represented with this feature.

The main reason the developers of CPLEX introduced indicator constraints was as an alternative to the big-M formulation, which can exhibit numerical problems when casual users choose liberally large big-M parameters. CPLEX handles these constraints specially in a manner that does not require any big-M style parameter and report that both numerical accuracy and computation times are substantially improved in many problems[1].

To allow experimentation, we have defined and implemented a transformation for converting arbitrary constraints in our language down to indicator constraints. We omit a formal definition as it is rather straightforward. Roughly, for every i^{th} disjunct in a disjunction, it replaces every inequality with an indicator constraint where the head is $y^i = 1$ and the body is the inequality itself, and the disjunction is replaced with a conjunction. Disjunctions are converted to DNF prior to applying this technique.

We have avoided adding implications in general to our object language because they raise additional complications, but we have added enough support to output indicator constraints to a format accepted by CPLEX.

3.3.3 Transformation of the top-level mathematical program

The transformations presented thus far can now be employed to define an overall constraint and program transformation.

Let $\Upsilon \vdash c \overset{\text{PROP}}{\longmapsto} c'$ mean within context Υ, constraint c is converted to c'. The definition simply employs the judgments we have detailed above for those forms requiring any transformation. Equations and inequalities are unaltered, and the procedure recurses on conjunctive and existential constraints. Our software allows selecting which specific transformation to use for the disjunctive constraint, and one can optionally perform a conversion to CNF or DNF on any constraint or Boolean expression.

Given this, transforming a mathematical program is now straightforward. Let $p \overset{\text{PROG}}{\longmapsto} p'$ represent a program transformation. The definition is

[1]Based on comments from the ILOG company's website. We are not aware of any published literature on indicator constraints.

$$\frac{\left\{\rho_j \overset{\text{RTYPE}}{\longmapsto} \rho'_j\right\}_{j=1}^{m} \qquad x_1 : \rho_1, \ldots, x_m : \rho_m \vdash c \overset{\text{PROP}}{\longmapsto} c'}{\max_{x_1:\rho_1,\ldots,x_m:\rho_m} \{e \mid c\} \overset{\text{PROG}}{\longmapsto} \max_{x_1:\rho'_1,\ldots,x_m:\rho'_m} \{e \mid c'\}} \tag{19}$$

Since the objective e must be of type `real`, it is already in MILP form and need not be transformed. The types and constraints are transformed using their respective procedures.

3.4 Implementation

We have implemented our object language as an embedded domain-specific language (EDSL) in OCaml, and all transformations are OCaml functions operating on this language's abstract syntax tree. All source code is freely available from the first author's website. The EDSL enables us to enhance our object language with functionality inherited from the host language. The primary benefits in our case are the following:

- Our object language does not have let bindings. Instead, we can use OCaml's `let`, which is useful for assigning names to model parameters, and long expressions and constraints.

- With OCaml as the meta-language, we can use OCaml functions to define macros over our object language. This is convenient for defining parameterized expressions and constraints. For example, in jobshop scheduling, there is a constraint that a job a must precede job b at a stage k, or vice versa. This can be succinctly written as `precedes a b k |/ precedes b a k`.

- Our object language does not include indexing, but we were able to define a set of functions that effectively do so at the meta-level. This is essential as no real-world MP model can be written without indexing.

We will see examples of each in the following section. It is surprising that the popular modeling languages such as GAMS and AMPL do not provide the first two benefits. Numerous complex expressions and constraints have to be repeated in full. CPLEX's C++ API does of course lead to similar capabilities with C++ serving as the meta-language.

The general strategy for indexing is to turn indexed variables into functions that generate the appropriate variable name. For instance, a variable $\gamma_{i,j}$ would be represented in OCaml

as

```
let gamma i j = ivar["gamma"; of_n i; of_n j]
```

where `ivar` is a helper function for creating indexed variables and `of_n` converts integers to strings. An expression like

```
let i = 1 in gamma i (i+1)
```

represents the object language variable `gamma_1_2`. This approach works because all indexing expressions are at the meta-language level and disappear by the time the object language compiler sees the program.

This is not to say that an EDSL is the optimal solution. For example, we do not get type-checking of object language programs when the OCaml compiler is invoked to compile EDSL code (instead, it occurs when the EDSL code is run). Techniques for extending EDSLs to provide more static safety can be provided by other techniques, such as generalized algebraic data types (GADTs), and is the subject of other work.

Once a program is specified in our EDSL, one of the various constraint transformations we have defined can be applied selectively or to the whole program. The transformed program, whether a pure MILP or an MILP enhanced with indicator constraints, can be printed to the industry-standard MPS format and the AMPL modeling language.

3.5 Results

We now present examples from chemical engineering and operations research that we model using the intuitive Boolean and disjunctive constraints supported by our software. We then analyze the programs automatically generated by applying the various transformations we have defined and will compare our solutions to both manually performed transformations, and a competing automated solution. We will look at the computational efficiency of different reformulations and compare the manual case to the automated case.

3.5.1 Experimental setup and performance metrics

We will be looking at the computational efficiency of different reformulations. To do this we will take the MP for our case studies and reformulate them to MILP form using different reformulation techniques. We will then solve the resulting MILP programs using ILOG's CPLEX solver—a widely used, efficient solver for, among other things, LP and MILP problems. We compare four transformation strategies:

- Three are our automated transformations of the big-M method, the convex-hull method, and our indicator constraint transformation. Only one input specification is needed for all three, namely, a version of the example implemented using our EDSL. Compiling the example with different options yields the three transformations.

- The fourth is CPLEX's Concert Technology. CPLEX offers a C++ API to their solver technology which allows a user to use C++ objects and overloaded operators to write down models in an intuitive manner. They provide Booleans and logical conditions over linear inequalities, which are automatically transformed into equivalent forms that use indicator constraints. The software is proprietary and their conversion to indicator constraints likely differs from the one we described in Section 3.3.2.

We do not compare to other software because either they do not support Boolean and disjunctive constraints or they call out to CPLEX making the comparison redundant. Mosel, another popular MP software, has an extension called Kalis that does support disjunctions, but it is a constraint programming solver using solution techniques unrelated to MILP algorithms.

For each of the above strategies, the metrics we will look at are:

- The number of continuous variables and constraints. These numbers give a rough picture of the potential computational difficulty of the program. Note that these counts treat indexed variables as distinct from each other, e.g. $x_1, \ldots, x_n$ counts as n variables rather than one; thus, they are not a good indication of the size of programs that human modelers *actually* manipulate. They are indicative only of algorithmic complexity, not modeling complexity.

- The number of discrete variables. This is especially relevant to computational complexity because solvers spend a large portion of their time branching on different possible values of discrete variables. In all of our examples, the only discrete variables are binary variables.

- CPU time needed for solving. This of course is the primary metric of interest. However, the other metrics give a better picture of what the transformations are actually doing.

All experiments were run on a machine running Linux 2.6.18 with 8GB of RAM, 4GB of swap space, and eight 2.6GHz Intel Xeon processors with 4MB caches.

3.5.2 Example: switched flow process

Figure 13 depicts a tank being filled by two pumps whose flow rates switch between two values depending on other requirements of the system. The tank is being emptied continuously at a rate of $F^{\text{out}} = 1.8$. Initially, the material level in the tank is $M_0 = 20.0$. The tank's maximum capacity $M^{\max} = 150.0$ and the material level should never fall below $M^{\min} = 10.0$ to avoid equipment damage.

Process α represents the first pump, which can be either on or off. When it is on, it provides material to the tank at rate 2.0. There is also an operating cost of 10.0 per unit time for running the pump. To avoid over-heating the pump, it is forbidden to continuously run it longer than 30.0 time units. There are no operating costs while it is off, but it must not be switched on again in less than 2.0 time units to allow it time to cool. When it is switched on again, if at all, a startup cost of 50.0 is incurred. Process β is similar, but it represents a pump that is always on, either at a high or low setting.

We wish to study how the material level changes over time and to understand the cost of running the system for $T^{\max} = 500.0$ time units. Our objective is to minimize cost. The first step is to formalize the problem. The most natural formulation involves disjunctive constraints and Boolean variables. The full model requires over two pages of constraints, so here we will present only the most instructive components relating to the transformations we are interested in.

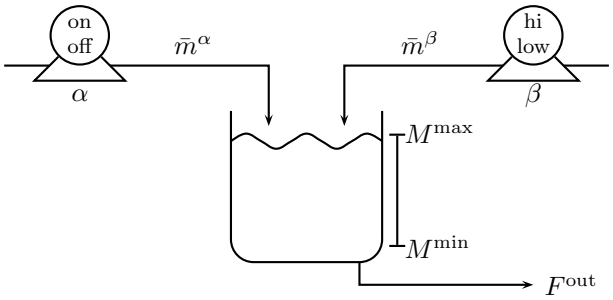


Figure 13: Schematic of switched flow process.

The first constraint we consider models the transitions of process α between its `on` and `off` modes:

$$\begin{bmatrix} Z^{\alpha}(\texttt{on},\texttt{off},i) \\ \hat{c}^{\alpha}(i) = 0.0 \\ \hat{r}^{\alpha}(i) = -R^{e}(i) \end{bmatrix} \vee \begin{bmatrix} Z^{\alpha}(\texttt{off},\texttt{on},i) \\ R^{e}(i) \geq 2.0 \\ \hat{c}^{\alpha}(i) = 50.0 \\ \hat{r}^{\alpha}\left(i\right) = -R^{e}\left(i\right) \end{bmatrix}$$
$$\vee \begin{bmatrix} YY^{\alpha}(i) \\ \hat{c}^{\alpha}(i) = 0.0 \\ \hat{r}^{\alpha}(i) = 0.0 \end{bmatrix} \qquad \forall i \in \mathbb{N}\backslash\{n\} \tag{20}$$

The set $\mathbb{N} = \{1, \ldots, n\}$ numbers the set of discrete events that occur during the execution of the system. The Boolean variable $Z^{\alpha}(q, q', i)$ indicates whether the process transitioned from mode q to q' at event i (e.g. from `on` to `off`). The variable $\hat{c}^{\alpha}(i)$ represents the cost incurred at event i. Some behaviors depend on the time elapsed since a previous event (e.g. the pump cannot be turned on again immediately after being turned off), we need some time-related bookkeeping: the clock variable $R^{e}(i)$ measures the time elapsed since the last event, and $\hat{r}^{\alpha}(i)$ is a helper variable that represents clock adjustment at the end of each event. For instance, the second disjunct states that if the pump transitions from `off` to `on` at event i, two time units must have elapsed since the last event, the cost incurred as a result will be 50.0 cost units, and the clock variable will be adjusted by exactly the opposite amount it currently is, i.e. it will be reset to zero. Finally, the Boolean variable $YY^{\alpha}(i)$ indicates whether event i is a "non-event", e.g. it switches from `on` to `on`. This

permits solutions where the pump changes its state fewer than, rather than *exactly*, n times. Altogether, this disjunction describes the transition behavior of process α: either the pump was turned off, resetting the clock and incurring no cost; or it was turned on after a period of time, resetting the clock, incurring cost; or nothing happened, incurring no cost and letting the clock run.

The second constraint involves only Boolean expressions and provides the definition of $YY^{\alpha}(i)$; namely, a "non-event" at event i for a process $a \in \{\alpha, \beta\}$ is when it transitions from `on` to `on` or from `off` to `off` (likewise with `hi` and `lo` for process β):

$$YY^{a}(i) \Leftrightarrow \bigvee_{q \in \mathbb{Q}^{a}} Z^{a}(q, q, i) \quad \forall i \in \mathbb{N} \backslash \{n\}, \forall a \in \{\alpha, \beta\} \tag{21}$$

where $\mathbb{Q}^{\alpha} = \{\texttt{on}, \texttt{off}\}$ and $\mathbb{Q}^{\beta} = \{\texttt{hi}, \texttt{lo}\}$.

Both of these constraints can be concisely written in our EDSL. Equation 20 is written as

```
(* disjunction over transitions of alpha *)
_CONJ(_N1, fun i->
   conj [ isTrue (goesFromTo Alpha On Off i);
          costJmp Alpha i ==^ lit 0.0;
          clockJmp R i ==^ neg (clock R Pe i); ]
|/ conj [ isTrue (goesFromTo Alpha Off On i);
          clock R Pe i >=^ lit 2.0;
          costJmp Alpha i ==^ lit 50.0;
          clockJmp R i ==^ neg (clock R Pe i); ]
|/ conj [ isTrue (isDummyTrans Alpha i);
          costJmp Alpha i ==^ lit 0.0;
          clockJmp R i ==^ lit 0.0; ]
);
```

and Equation 21 is written as

```
(* definition of isDummyTrans *)
_CONJ(_N1,fun i-> isTrue ( isDummyTrans Alpha i <==>
  (goesFromTo Alpha On On i ||^ goesFromTo Alpha Off Off i) ));
_CONJ(_N1,fun i-> isTrue ( isDummyTrans Beta i <==>
  (goesFromTo Beta Hi Hi i ||^ goesFromTo Beta Lo Lo i) ));
```

The EDSL is fairly straightforward, and we can see that the code coincides nicely with the mathematical notation. In most cases, we have appended a caret (`^`) to standard

Table 2: Switched flow process: Comparison of automatic reformulations. IC, BM and CH refer to the indicator constraint, big-M, and convex-hull transformations as implemented by our software.

Method	#vars (#binary)	#constr. (#IC)	solve time (sec)
Concert	1061 (874)	1080 (718)	36.85
IC	477 (291)	1001 (438)	11.60
BM	477 (291)	1198	3.37
CH	1194 (631)	2747	1.09

OCaml relational and arithmetic operators to denote the corresponding MP operator (`+^`, `<=^`, etc.). We distinguish Boolean `and` and `or` (`&&^` and `||^`) from the propositional operators $\wedge$ and $\vee$ (`/|` and `|/`). Because capitalized names are reserved for modules and constructors in OCaml, some variable and function names must be prepended with an underscore. Furthermore, literals must be lifted to the object level using `lit`. The function `conj` generates a conjunction from a list of constraints while `_CONJ` generates a conjunction from an index set and an indexed expression.

The performance metrics for experiments on this example are in Table 2. The main outcome is that the methods perform largely as expected: tighter formulations are solved faster. Indeed, convex-hull is the fastest formulation despite generating the largest number of constraints. As expected, the big-M method uses the same number of binary variables as the indicator constraint transformation, but needs a larger number of constraints because it handles equality constraints as a pair of inequalities, while the indicator constraint transformation handles equalities directly. Curiously, the Concert formulation introduces more binary variables than the convex hull method, more indicator constraints than our indicator constraint transformation, and is the slowest. Overall, we can see that for this example our transformations perform reasonable reformulations that in fact outperform an existing automated transformation provided by a state-of-the-art solver.

3.5.3 Example: strip packing

It is however known that the convex-hull method can perform poorly on problems with a large number of disjunctions. The *strip packing* problem from operations research involves packing n rectangles without rotation or overlap into a strip of width W that is unbounded

to the right. The goal is to minimize the length of the strip needed to pack the rectangles. This is a frequently studied problem and we have available reformulations done manually by experts, which allows us to compare our automatically generated programs with expertly generated ones.

The mathematical model of the strip packing problem is

$$\begin{aligned}
\min \quad & lt && \text{(22a)}\\
\text{s.t.} \quad & lt \geq x_i + L_i \quad \forall i \in \mathbb{N} && \text{(22b)}\\
& [x_i + L_i \leq x_j] \\
& \vee [x_j + L_j \leq x_i] \\
& \vee [y_i - H_i \geq y_j] \\
& \vee [y_j - H_j \geq y_i] \quad \forall i, j \in \mathbb{N}, i < j && \text{(22c)}\\
& 0 \leq x_i \leq UB - L_i \quad \forall i \in \mathbb{N} && \text{(22d)}\\
& H_i \leq y_i \leq W \quad \forall i \in \mathbb{N} && \text{(22e)}
\end{aligned}$$

where $\mathbb{N} = \{1, \ldots, n\}$. This formulation tracks the (x, y)-position of the top left corner of each rectangle and relates them via constraints involving the rectangle lengths and heights, L_i and H_i. The first constraint states that the optimal length is greater than the rightmost side of each rectangle; the disjunctive constraint states that for a rectangle i, another rectangle j must either be to the right, to the left, below, or above of it (in a non-overlapping manner); and the final constraints are just some bounds on the positions. The parameter UB is an upper bound on the optimal value; in our experiments, we set this equal to the sum of the rectangle lengths.

For our experiments, we implemented the MP form of strip packing with our EDSL and compared it to reformulations manually performed by an expert of both the big-M and convex-hull methods. The manual reformulations were taken from [58], and we used them verbatim, with no modifications. We then ran the reformulations on a medium problem consisting of 12 rectangles and a large problem consisting of 21 rectangles. The results are in Table 3 and Table 4, respectively.

Table 3: Strip packing (12 rectangles): Expert vs. automatic reformulations.

Method	#vars (#binary)	#constr. (#IC)	solve time (sec)
IC	289 (264)	342 (264)	1.83
BM	289 (264)	342	1.22
CH	1345 (264)	2718	168.38
BM, expert	289 (264)	342	1.82
CH, expert	1345 (264)	1662	149.57

Table 4: Strip packing (21 rectangles): Expert vs. automatic reformulations.

Method	#vars (#binary)	#constr. (#IC)	solve time (sec)
IC	883 (840)	1071 (840)	24.44
BM	883 (840)	1071	55.01
CH	4243 (840)	8631	991.68
BM, expert	883 (840)	1071	29.56
CH, expert	4243 (840)	5271	$\geq$ 3600.00

The results show that convex-hull is indeed not the optimal solution technique in all scenarios. The number of constraints and variables outweighs any benefits from having a tight formulation per disjunction. Also, we can see that the automatic versions of the big-M and convex-hull transformations are on par with the expertly coded versions. The number of binary variables is equal across all methods because they all introduce one binary variable per disjunct, and there are no Boolean variables in the source program. Many of the numbers are identical between the expertly coded and automated versions, as expected with the simple program structure of strip packing. Also, the expertly coded convex-hull method contains fewer constraints because the expert is able to reason that some constraints are redundant given their bounds, e.g. $0 * y \leq x_i$ is unnecessary for constraining x_i if it is already declared that $x_i \in \mathbb{R}^+$.

In general, it is hard to tell a priori which methods will work well on a given program, so it is useful to have a tool such as ours that enables experimentation without the manual overhead. In fact, anecdotal evidence suggests that once the object language has been properly formalized, adding reformulations is quite easy, so there is a lower barrier to trying new ideas.

3.6 Related and future work

Egon Balas first described the convex-hull method in a technical report [5], which was made available in published form much later [4]. The theory presented there has had significant impact on MILP algorithms. Although Balas acknowledged that disjunctive constraints are useful for modeling, the focus has been on the insights they provide to more computationally efficient formulations. Thus, those working on MP theory have had little motivation to automate transformations and have not considered the differences arising from programs written in non-matrix forms.

Raman and Grossmann [54] popularized this method amongst the chemical processing industry and demonstrated that complex real-world problems could be modeled effectively. They also included the use of Boolean constraints, and provided a method for tying these to disjunctive constraints.

Vecchietti and Grossmann [65] describe an implementation of this alternative formulation with similar goals to this work in a software called LogMIP, implemented as an extension of the GAMS language. They support the convex-hull method, but it is not difficult to find examples where the software provides erroneous answers, and the semantics of the input language are rather unclear. For example, the disjunction

$$[x_1 + x_2 \leq 5.0] \vee [x_1 + x_2 \geq 10.0]$$

has to be written the following way:

```
equations eq1,eq2,eq3;
eq1.. x1 + x2 =l= 5.0;
eq2.. x1 + x2 =g= 10.0;
eq3.. y =g= 0;

x1.lo = 1.0;
x1.up = 100.0;

$ONTEXT BEGIN LOGMIP
```

```
DISJUNCTION D1;


D1 IS
IF Y THEN
eq1;
ELSE
eq2;
ENDIF;
$OFFTEXT END LOGMIP
```

The initial lines are pure GAMS, and the LogMIP extensions must be written within GAMS comments, enclosed by the `$ONTEXT` and `$OFFTEXT` lines. The disjunction is expressed as an `IF ... THEN ... ELSE ...` statement with the clauses defined as `eq1` and `eq2`. These are the names of equations defined above in pure GAMS. However, the initial definitions of `eq1` and `eq2` are GAMS syntax for requiring both to be true. Thus local reasoning is broken. The meaning of the initial lines is entirely changed by the addition of the LogMIP declaration later in the program. Furthermore, using an if-then-else statement to represent a disjunction appears a bit more like indicator constraints, not the normal disjunctive constraints on which so much of MILP theory is based. Finally, we found that an incorrect answer is returned when we ran this program. It appears to be because some arbitrary bounds are provided for `x2`, which is not specified with bounds in the input. It is our hope that the theory developed in this work can be employed as a foundation for future development of LogMIP.

Sawaya [58] and Liberti [35] discuss many more transformations besides the big-M and convex-hull methods. Many are related to forms other than disjunctive constraints and so we feel our syntactic formulation can have wider benefits. Nemhauser and Wolsey [46] and others discuss the importance of cuts, which our syntactic foundation should be able to support.

Hooker [27] discusses the promising idea of employing constraint programming (CP) techniques to solve mathematical programs. One of the challenges for these efforts has

been that traditional MP theory does not allow referring to constraints as objects which is essential to CP. Our syntactic formulation immediately provides this.

We have compared our software to CPLEX[2], which is considered the state-of-the-art MILP solver. In addition, with respect to the language features we are considering, its API is the most expressive. It supports Booleans and disjunctive constraints to the full generality that we do. It also provides a syntactic conversion of these (to indicator constraints) and was thus the most appropriate tool for comparing our transformations to. Note however that CPLEX has numerous other features making it an effective algorithm. Our goal is to supplement those capabilities with operations benefiting from a syntactic perspective.

There are other works that focus specifically on language design. The most widely used are GAMS [10], AMPL [21], Mosel [13], and OPL [64]. [30] provides a comprehensive overview. All these support indexing, an essential requirement of any good MP language. It is interesting that although these are the leading languages, they have limited or no support for important features such as Booleans and disjunctive constraints. Although our goal in this work was not to provide a superior object language, we believe our use of formal programming language methods can lead to better languages.

3.7 Conclusion

We have demonstrated the promise of the formal approach introduced by Tyles by (i) adding a new transformation, the big-M method, and by (ii) showing that it can indeed be connected to the real world by providing an end-to-end implementation. Hopefully we have convinced the reader that programming language methodologies can address important challenges in mathematical programming.

Automation of course has many benefits. Programs are less likely to contain errors, and good formulations will no longer be restricted to experts familiar with the theory. Also programs can be maintained in the more compact and intuitive form, easing their development. As we implement more transformations, compiler optimizations can be considered.

[2] http://www.ilog.com

For instance, we mentioned that sometimes the big-M method is better than the convex-hull method. Automating these decisions should lead to programs that can be solved faster than what manual formulation is likely to achieve. In short, all the usual benefits of programming language design can benefit the practice of mathematical programming, which has not previously employed formal methods in their software implementations. Our work initiates a framework for systematically supporting more elegant constraint forms as well automating the many more transformations [35] widely used in mathematical programming.

CHAPTER IV

A SYNTACTIC THEORY OF MACHINE LEARNING

While the research community has recognized the need to study probabilistic programming languages—which aid users in specifying and solving stochastic models—there has not been enough work on languages that embrace all facets of machine learning. In particular, there is little work on languages that incorporate *optimization* in addition to probability. In this chapter, we help fill this void by combining the probabilistic language from Chapter 2 and the optimization language from Chapter 3 into a single syntactic theory for machine learning, implemented using the Coq proof assistant. We demonstrate the applicability of the theory by mechanizing techniques from different corners of machine learning.

4.1 Introduction

There has been considerable interest in designing and implementing *probabilistic programming languages*, which aim to simplify the task of specifying and performing inference on stochastic models. However, these languages only represent the *probabilistic* account of machine learning. Notably, they do not provide a primitive for *optimization* problems. This is quite unfortunate: many techniques in machine learning make key use of optimization in their formulations, sometimes omitting any mention of probability altogether! Ideally, we would extend these probabilistic languages to incorporate optimization, which would allow users to write programs that use either or both primitives. This is also a morally satisfying property for any "language for machine learning". Programs which combine probability and optimization are able to *intrinsically* represent learning problems, rather than having the notion of learning be imposed by an entity *external to* the program, as we will see with maximum likelihood estimation.

The primary contribution of this chapter is a unified syntactic theory of machine learning that incorporates *both* probability and optimization. We achieve this by unifying the formal languages for probability and optimization introduced in the previous chapters, providing

primitive constructs for both in our formalization. We use a *type-theoretic* approach to formalizing the necessary mathematical objects, which is the key step that enables this unification. We show that our formalization is expressive, promotes correctness, and enables mechanizing useful machine learning techniques:

- **Expressive.** We show that the formalization captures multiple facets of machine learning by mechanizing an example that focuses on optimization as well as an example that combines probability and optimization. The programs and rewrite theorems implementing the examples are in a nearly one-to-one correspondence with their respective mathematical statements, demonstrating a low semantic gap between our language and established mathematical notation.

- **Promotes correctness.** Our theory promotes correctness to a further degree than related works. We achieve this by (i) building upon formally defined languages for probability and optimization, which helps reasoning about programs, and (ii) expressing program rewrites as theorems, which provides a path to fully verified algorithm derivations.

- **Enables mechanization of important techniques.** We demonstrate the utility of the theory by using it to mechanize (i) the *big-M transformation of* L_0 *regularization* for solving L_0 *support vector machines* and (ii) the *expectation maximization* principle for solving *maximum likelihood estimation* problems.

We implement these ideas in the Coq proof assistant [6]. Coq is based on a foundational type theory, and this is an important feature of our approach. We reflect on this choice in the conclusion, after describing our examples.

4.2 Implementing our language as a Coq *theory*

A *proof assistant* is software that assists users in the construction of machine-checked proofs. The Coq system consists of a term language, named Gallina, for defining mathematical objects and proofs over those objects, as well as a command language for interactively constructing proof objects. The term language is based on the *Calculus of Constructions*,

a dependently typed version of the lambda calculus. Such dependent type theories were designed to provide syntactic foundations for mechanizing mathematics, making them a viable vehicle for mechanizing machine learning. In fact, we have designed the languages presented in previous chapters to be compatible with Gallina. Furthermore, these theories also have a computational interpretation, which means they are suitable as programming languages in their own right. This ability to represent mathematics and code in the same term language fits our needs perfectly, because we will need to iteratively refine a declarative mathematical problem statement to an equivalent program that is computational.

Proving a theorem proceeds as follows. First, the user defines any mathematical objects necessary for stating the theorem. These definitions may come from "traditional" mathematics, such as constructions of real numbers, or from programming, such as data structures like lists or balanced binary trees. Next, the user states a theorem of interest, which starts an interactive theorem proving mode. In this mode, the current *proof state* is always displayed, listing the current set of assumptions and the current set of goals and subgoals left to be proved; to start, the set of assumptions is empty and the set of remaining goals consists only of the main theorem. The user can then enter commands to manipulate the current proof state, such as applying lemmas to reduce complex goals into a set of simpler subgoals, until only trivial subgoals remain.

We borrow this mechanism for our task of expressing and solving machine learning problems. We define constructs for probability distributions and optimization problems, in a fashion similar to definitions from the previous chapters. We achieve algorithm derivation by first asserting the existence of an object that satisfies the specification of a desired learning problem and then building a constructive proof that witnesses the object's existence. This witness will be an algorithm that computes a solution to the learning problem, as we will see in our examples. Finally, we can use Coq's program extraction capability on the witness to produce an executable OCaml version of the algorithm. We elaborate on these ideas in the following sections.

4.3 Machine learning as a Coq *theory*

As mentioned above, Coq's term language Gallina is based on the well-established lambda calculus and supports standard features such as function abstraction and application, let-expressions, and recursive datatypes. Furthermore, Coq's standard library defines theories for many common types, including Booleans, natural numbers, and real numbers. All that remains is to define new operators related to probability and optimization. We start with the notion of *stocked spaces* from Chapter 2. We do this by writing

```
Axiom stocked_spec : forall (A : Type), Prop.
```

which postulates a new predicate `stocked_spec` which, when given a type `A`, returns a proposition that is intended to hold when `A` represents a stocked space. We will not need to make use of the actual definition of a space being stocked, so we use the `Axiom` keyword instead of the `Definition` keyword, which allows us to postulate a new uninterpreted function with a specific signature without needing to supply an implementation. We hook this into Coq's typeclass mechanism with

```
Class stocked (A : Type) := {
  stocked_H0 : stocked_spec A
}.
```

which defines a new parametric record type `stocked` with a type parameter `A` and a field `stocked_H0` that holds a proof of the proposition `stocked_spec A`, *i.e.* that `A` is indeed a stocked space. We do this so that we can leverage *typeclass resolution*, which is a mechanism that uses proof search to automatically populate such records where needed. For example, the declaration

```
Axiom dist : forall A `{stocked A}, Type.
```

introduces the type constructor `dist` that, when given a type `A` and a proof that `A` is stocked, returns a type representing distributions over `A`. The tick mark (`) notation can be considered as shorthand for `{_ : stocked A}`. The curly braces denote an argument that will be inferred by Coq (by either typeclass resolution or the type inference algorithm) and

does not need to be supplied by the programmer. So, although `dist` takes two arguments, the user only needs to write `dist bool` to specify the type of Boolean distributions. The resolution is informed by a database of *typeclass instances*, which are lemmas about when one can deduce the existence of a record with the type in question. For example, the code

```
Instance stocked_R : stocked R.
Proof. admit. Defined.

Instance stocked_prod A B `{stocked A} `{stocked B} : stocked (A * B).
Proof. admit. Defined.
```

first asserts that the space of real numbers `R` is stocked and then asserts that the product space `A * B` of two stocked spaces `A` and `B` is itself stocked. With these two facts, the system can infer that two-dimensional Euclidean space `R * R` is stocked, which would arise if the user were to write `dist (R * R)` in a program. The proofs of these lemmas have been *admitted* by the programmer, instructing the system to accept them without proof, analogous to the `Axiom` mechanism. This step is necessary so long as `stocked_spec` is stated as an axiom, but the user can at any time go back and start replacing admitted definitions and proofs with ones written from first principles, achieving an assumption-free verified codebase.

In addition to a type for distributions, we also need ways to build the distributions themselves. We use the same three combinators from before:

```
Axiom random : dist {x : R | 0 < x < 1}.
Axiom ret : forall {A} `{stocked A}, A -> dist A.
Axiom bind : forall {A B} `{stocked A} `{stocked B},
  dist A -> (A -> dist B) -> dist B.
```

We use a *subset type* to give `random` a more informative type of a distribution on the unit interval, rather than just a distribution on the entire real line. The functions `ret` and `bind` refer to the return and var constructs from before and correspond to the monadic return

and bind operations in the probability monad. The types and proofs of stockedness are inferred automatically. We use Coq's `Notation` mechanism to define the syntactic sugar

```
Notation "'var' x ~ P 'in' Q" := (bind P (fun x => Q))
  (at level 100, x ident, right associativity).
```

This allows use to write distributions using the more familiar notation from before:

```
Program Definition std_normal : dist R :=
  var u ~ random in
  var v ~ random in
    ret (sqrt (-2 * ln u) * cos (2 * PI * v)).


Definition mix {A} `{stocked A} (p : R) (P1 P2 : dist A) : dist A :=
  var z ~ flip p in
    if z then P1 else P2.
```

The first example defines the standard normal distribution as before. It makes use of the `Program Definition` facility to seamlessly insert projections from `{x : R | 0 < x < 1}` to `R` as necessary, which are needed because `u` and `v` have the former type while the standard library functions `sqrt`, `ln`, and `cos` are `R`-valued. The second example defines a generic mixture combinator. Note that it requires a proof that `A` is stocked just to be able to write `dist A` in the signature.

We also define an operator for computing the probability density function of a distribution. We again use the typeclass mechanism, not only for automatically producing *proofs*, as we did with `stocked`, but also for producing *code*. As before, we start with a specification for density functions, stated axiomatically:

```
Axiom is_pdf : forall {A} `{stocked A} (P : dist A) (f : A -> R), Prop.
```

When given a distribution `P` and a function `f`, `is_pdf` returns a proposition that is intended to hold when `f` is a PDF of `P`. This is used in the `has_pdf` typeclass,

```
Class has_pdf {A} `{stocked A} (P : dist A) := {
  pdf : A -> R;
  has_pdf_H0 : is_pdf P pdf
}.
```

which defines a record type with a field `pdf` that holds a function and a field `has_pdf_H0` that holds a proof that the function is a PDF of `P`. This record is automatically populated when `P` has a PDF that is deducible from the typeclass instances. Instances are used to encode the PDF calculation rules from Section 2.5 in a more or less one-to-one fashion. For example, rule P-LINEAR can be written

```
Instance has_pdf_translate
  {A} `{stocked A} (P : dist A) (f : A -> R) (c : R)
  {_ : has_pdf (var x ~ P in ret (f x))}
: has_pdf (var x ~ P in ret (f x + c)) := {
    pdf := fun x => pdf (var x ~ P in ret (f x)) (x - c)
}.
Proof. admit. Defined.
```

Essentially, the constant `has_pdf_translate` is evidence that distributions of the form `var x ~ P in ret (f x + c)` have a PDF when `var x ~ P in ret (f x)` has a PDF. The premises of P-LINEAR appear as arguments to `has_pdf_translate` and the conclusion of P-LINEAR appears in the return type and record contents of `has_pdf_translate`. The proof that the function is indeed a PDF has been admitted. We can also specify PDFs for specific distributions, as needed:

```
Definition phi x := exp (- x*x / 2) / sqrt (2 * PI).
Instance has_pdf_std_normal : has_pdf std_normal := { pdf := phi }.
  Proof. admit. Defined.
```

These instances are combined automatically during typeclass resolution, able to resolve programs such as `pdf (var x ~ std_normal in (x + 5))`. The resulting PDF terms can

further be extracted to OCaml programs as long as implementations are provided for the real number operations. The current implementation uses floating point.

Finally, we declare an operator for optimization, based on the signature and semantics given by Agarwal:

```
Axiom min : forall {A} (f : A -> R) (c : A -> Prop), option R.
```

When given an objective function `f` and an optimization constraint `c`, the operator `min` returns a value of type `option R`, which can take one of two forms: `None` for when no minimum value exists, or `Some v` for when a minimum value `v` exists. The minimization ranges over some space `A`. Our introductory disjunctive constraint optimization example can now be written

```
min (fun (x1,x2) => x1 - x2)
    (fun (x1,x2) =>    (x1 >= 1 /\ x2 >= 1 /\ x1 + x2 <= 5)
                    \/ (5 <= x1 <= 8 /\ 4 <= x2 <= 7) )
```

using Coq's logical connectives for conjunction (`/\`) and disjunction (`\/`). For presentational simplicity, we write examples in a hypothetical version of Gallina that allows irrefutable pattern matching (destructuring bind) in a function's argument list.

4.3.1 Semantics

We employ a *shallow embedding* of the probability monad so that we can use Gallina as our object language and thus program directly with the semantics. All that remains to achieve a fully defined semantics for this syntactic theory (based only on axioms from classical mathematics) is to provide definitions for the few primitives that we have postulated using the `Axiom` keyword or the `admit` tactic; these are `stocked_spec`, admitted instances of the `stocked` typeclass, `dist`, `random`, `ret`, `bind`, `is_pdf`, and `min`. The other operations (numeric, Boolean, and propositional operations) are defined from first principles in the Coq standard library.

The semantics of these constructs are nearly identical to the semantics given in previous chapters, so we give just an informal treatment here, mainly referring back to previous

definitions and addressing some minor differences. The phrase `stocked_spec A` is defined as the proposition that `A` is a measurable space (there exists a suitable σ-algebra) and that there exists a measure on `A` (a function of type $(A \to \mathbb{R}^+_\infty) \to \mathbb{R}^+_\infty$ that satisfies the measure laws), which gives us the stock integral. We achieve the stock integral by the standard construction for the integral over arbitrary real functions in terms of the integral over nonnegative real functions. The instances of `stocked` for reals, integers, Booleans, and pairs have the semantics given in Section 2.3.3. The semantics for `dist`, `random`, `ret`, and `bind` are entirely standard and are given by the probability monad, defined in Section 2.3.4. The only minor difference is that here the type constructor `dist` requires the space to be stocked, which is stronger than necessary (strictly speaking, the space need only be measurable); this choice is made merely for convenience and could be refactored into a more precise hierarchy in a straightforward manner.

The phrase `is_pdf P f` is defined as in Section 2.3.3: the proposition that, for any measurable set X, the probability of X under the distribution `P` is equal to the stock integral of the function `f` on X. And, in contrast to Tyles, our minimization operator `min` is a higher-order function instead of a separate variable binding construct. In essence, we reuse the variable binding capability provided by lambda functions, as is usual in formalizations of mathematics using type theories such as the Calculus of Constructions. The phrase `min f c` represents the minimization of the objective function `f` subject to the optimization constraint `c`. As in Tyles, the operator returns a value of type `option R`, which is `Some r` when a value `r` exists such that no setting of the program variable respecting the constraint attains a smaller value, or is `None` when no such value exists. In fact, it is straightforward to define a version of `min` with a stronger specification, based only on axioms from classical mathematics, from which `min` itself can be defined:

```
Require Import ClassicalEpsilon.
Definition is_min {A} (f : A -> R) (c : A -> Prop) (r : R) : Prop :=
  forall (x : A), c x -> f x >= r.
Definition min' {A} (f : A -> R) (c : A -> Prop)
  : {r | is_min f c r} + {~ exists r, is_min f c r}.
```

```
Proof.
  destruct (excluded_middle_informative (exists r, is_min f c r)) as [y|n].
    pose (constructive_indefinite_description _ y); intuition.
    intuition.
Defined.
```

This operator `min'` (morally) returns an `option R`, decorated with proofs showing that either the returned value indeed attains the minimum value (in the `Some r` case) or that there does not exist a minimum value (in the `None` case). As in Tyles, we do not make a distinction between unbounded and infeasible optimization problems.

4.4 The big-M method for L_0 regularization

4.4.1 Sparsity in support vector machines: the L_0-SVM formulation

The *support vector machine* (SVM) is a powerful and widely used technique from machine learning for solving classification problems [12]. Various extensions of SVMs have been proposed and studied; one of these is the L_0-SVM, which augments the SVM formulation to induce solution sparsity. The L_0-SVM starts with the standard SVM primal formulation,

$$\min_{\mathbf{w},b,\xi} \ \frac{1}{2}\|\mathbf{w}\|_2 + c\|\xi\|_1 \quad \text{s.t.} \quad \begin{array}{l} \mathbf{w} \in \mathbb{R}^d \quad b \in \mathbb{R} \quad \xi \in \mathbb{R}^n_+ \\ \mathbf{Y}(\mathbf{X}\mathbf{w} - b) + \xi \geq \mathbf{1} \end{array} \tag{23}$$

where $\mathbb{R}_+$ is the nonnegative reals, $n \in \mathbb{N}$ is the number of data points, $d \in \mathbb{N}$ is the number of features, $\mathbf{y} \in \{-1, +1\}^n$ contains the labels, $\mathbf{Y} = \operatorname{diag}(\mathbf{y}) \in \mathbb{R}^{n \times n}$, $\mathbf{X} \in \mathbb{R}^{n \times d}$ contains the dataset, and $c \in \mathbb{R}_+$ controls how much to penalize the classification errors, as measured by the slack variables ξ. The bias term b and the weight vector $\mathbf{w}$ specify the separating hyperplane. Next, the L_0-SVM adds an L_0 penalty term on $\mathbf{w}$ to induce sparsity in $\mathbf{w}$,

$$\min_{\mathbf{w},b,\xi} \ \frac{1}{2}\|\mathbf{w}\|_2 + c\|\xi\|_1 + a\|\mathbf{w}\|_0 \quad \text{s.t.} \quad \begin{array}{l} \mathbf{w} \in \mathbb{R}^d \quad b \in \mathbb{R} \quad \xi \in \mathbb{R}^n_+ \\ \mathbf{Y}(\mathbf{X}\mathbf{w} - b) + \xi \geq \mathbf{1} \end{array} \tag{24}$$

where $a \in \mathbb{R}_+$ controls how much to pay attention to solution sparsity. Both a and c are selected by the user and can be used to explore different levels of sparsity and toleration of errors. We can easily state the L_0-SVM in our Coq formalization. We first define vectors of length n as functions indexed by the set $\{0, \ldots, n-1\}$,

```
Definition fin (n : nat) := {k : nat | k < n}.
Definition vec (A : Type) (n : nat) := fin n -> A.
Notation "A ^ n" := (vec A n).
```

where `nat` is the type for natural numbers defined in the Coq standard library. Indexing into vectors is just ordinary function application. We also declare norm operations and dot product, stated axiomatically for convenience:

```
Axioms L0 L1 L2 : forall {n}, R^n -> R.
Axiom dot : forall {n}, R^n -> R^n -> R.
```

Implementations for these operations can be provided at a later stage if necessary. The Coq version of L_0-SVM is nearly identical to its mathematical statement, with the minor cosmetic difference of using `forall`-propositions ranging over vector indices to encode the vector inequalities:

```
Program Definition L0_SVM
  {n d : nat}    (* number of points, dimensions *)
  (x : (R^d)^n)  (* the feature vectors *)
  (y : R^n)      (* the labels, from {-1,+1} *)
  (a c : R)      (* penalty tradeoff coefficients *)
:=
  min (
    fun (w,b,k) : R^d * R * R^n =>
      c * L1 k + L2 w / 2 + a * L0 w
  )(
    fun (w,b,k) =>
      (forall i, k i >= 0) /\
      (forall i, y i * (dot w (x i) - b) + k i >= 1)
  ).
```

4.4.2 Solving L_0-SVM with mixed-integer SVMs

Unfortunately, L_0-SVMs cannot be solved efficiently in their given form and must be converted to a formulation that deals directly with the discrete and jumpy nature of a vector component being zero or nonzero. To accomplish this, we modify the big-M transformation for disjunctive constraints to handle the discrete choice inherent in L_0 penalty terms, which occur in objective functions. We first illustrate this new transformation using a simple example before presenting the more general version. Consider the following optimization problem,

$$\min_{\mathbf{x}} \; f(\mathbf{x}) + \|\mathbf{g}(\mathbf{x})\|_0 \quad \text{s.t.} \quad \mathbf{x} \in \mathbb{R}^d \tag{25}$$

where $d, k \in \mathbb{N}$, $f : \mathbb{R}^d \to \mathbb{R}$ is a cost function, and $\mathbf{g} : \mathbb{R}^d \to \mathbb{R}^k$ is a vector-valued penalty function. We take the L_0 norm to induce sparsity in $\mathbf{g}(\mathbf{x})$. We can rewrite this by introducing a new optimization variable,

$$\min_{\mathbf{x},\mathbf{z}} \; f(\mathbf{x}) + \|\mathbf{z}\|_1 \qquad \text{s.t.} \quad \begin{array}{l} \mathbf{x} \in \mathbb{R}^d \qquad \mathbf{z} \in \{0,1\}^k \\ M_L \cdot \mathbf{z} \;\leq\; \mathbf{g}(\mathbf{x}) \;\leq\; M_U \cdot \mathbf{z} \end{array} \tag{26}$$

where M_L and M_U are lower and upper bounds on the value that any component of $\mathbf{g}(\mathbf{x})$ can take. This reformulation works by having each component of the binary vector $\mathbf{z}$ model whether or not the corresponding component of $\mathbf{g}(\mathbf{x})$ is zero. The key issue is to ensure that components of $\mathbf{g}(\mathbf{x})$ do in fact get set to zero when their corresponding components in $\mathbf{z}$ are set to zero. This is accomplished by the big-M (vector) inequality, which for each index i reduces to $0 \leq \mathbf{g}_i(\mathbf{x}) \leq 0$ when $\mathbf{z}_i = 0$, forcing $\mathbf{g}_i(\mathbf{x})$ to be zero, and to $M_L \leq \mathbf{g}_i(\mathbf{x}) \leq M_U$ when $\mathbf{z}_i = 1$, which is trivially satisfied and thus does not have an effect on $\mathbf{g}_i(\mathbf{x})$.

A basic translation of this intuition to Coq is straightforward. We first introduce a refinement of the reals to use as the type of binary variables:

```
Definition Bin := {x : R | x = 0 \/ x = 1}.

Definition LB  := -1000000.

Definition UB  :=  1000000.
```

We hard-code the lower and upper bounds for simplicity of presentation. We can now state the reformulation as an equality theorem.

```
Program Axiom L0_bigM :
  forall (A : Type) (n : nat) (r : R)
         (f : A -> R) (g : A -> R^n) (c : A -> Prop),
  min (fun x : A => f x + r * L0 (g x))
      (fun x => c x)
  =
  min (fun (x,z) : A * Bin^n => f x + r * L1 z)
      (fun (x,z) => c x /\ forall i, LB * z i <= (g x) i <= UB * z i).
```

This statement generalizes the example by permitting the minimization to range over an arbitrary space `A` and by accounting for any existing optimization constraint `c`.

Now we show how to borrow the theorem proving mechanism to perform algorithm derivation. First, we state a specially formulated existence theorem:

```
Theorem MI_SVM :
  forall {n d} (x : (R^d)^n) (y : R^n) (a c : R),
    {ans : option R | ans = L0_SVM x y a c}.
```

This proposes that for any given inputs, there is an answer (of type `option R`) that is the same as the result given by the L_0-SVM on those inputs (*i.e.* the set of such answers is non-empty); the process of proving this theorem amounts to constructing a term that represents such an answer. This term will be the reformulation of the original specification, *i.e.* of L_0-SVM. The following proof script does the actual algorithm derivation:

```
Proof.
  intros; unfold L0_SVM.      (* line 1 *)
  rewrite L0_bigM; simpl.     (* line 2 *)
  eapply exist; reflexivity. (* line 3 *)
Defined.
```

Lines 1 and 3 are fairly clerical in nature; the main action happens in line 2, where we apply our reformulation `L0_bigM` using the `rewrite` tactic. Figures 14 and 15 show the proof

```
1 subgoal

  n : nat
  d : nat
  x : (R ^ d) ^ n
  y : R ^ n
  a : R
  c : R
  ============================
   {ans : option R |
   ans =
   min
     (fun (k,b,w) : R ^ n * R * R ^ d =>
      c * L1 k + / 2 * L2 w + a * L0 w)
     (fun (k,b,w) : R ^ n * R * R ^ d =>
      (forall i : fin n, k i >= 0) /\
      (forall i : fin n,
       y i * (dot w (x i) - b) + k i >= 1))}
```

Figure 14: Proof state before applying the big-M rewrite.

```
1 subgoal

  n : nat
  d : nat
  x : (R ^ d) ^ n
  y : R ^ n
  a : R
  c : R
  ============================
   {ans : option R |
   ans =
   min
     (fun (k,b,w,z) : R ^ n * R * R ^ d * Bin ^ d =>
      c * L1 k + / 2 * L2 w +
      a * L1 (fun i : fin d => `(z i)))
     (fun (k,b,w,z) : R ^ n * R * R ^ d * Bin ^ d =>
      ((forall i : fin n, k i >= 0) /\
       (forall i : fin n,
        y i * (dot w (x i) - b) + k i >= 1)) /\
      (forall i : fin d,
       LB * `(z i) <= w i <= UB * `(z i)))}
```

Figure 15: Proof state after applying the big-M rewrite.

state displayed by the system before and after line 2 is executed. Again, for reasons of clarity in presentation, the output has been modified in the style indicated before (using pattern matching in argument lists). The system successfully unifies and instantiates the different pieces of the `LO_bigM` reformulation with respect to the subject of the reformulation, `LO_SVM`. The resulting formulation is exactly the *mixed integer SVM* of Guan et al. [25], albeit reached via a different series of transformations.

4.5 Expectation maximization for maximum likelihood estimation

4.5.1 The MLE formulation of parameter estimation

Maximum likelihood estimation (MLE) is a classic technique from statistics for performing *parameter estimation*, the task of finding parameters for a parameterized stochastic model that "best explain" observed data. Recall from Section 2.2.3 that MLE takes the "best" parameters to be the ones which maximize the likelihood function of the model,

$$\theta^* = \arg\max_{\theta} \; f(x;\theta) \tag{27}$$

where x is some observed data and f is the parameterized PDF of the model. The function $L(\theta) := f(x;\theta)$ is known as the likelihood function. It is possible to derive closed-form solutions of θ^* for simple models. Take for example a dataset of numbers $x_1, \ldots, x_n$ which we believe are drawn independently and identically from a normal distribution, whose parameters (mean μ and variance σ^2) we would like to estimate. With some calculus we can show that $\mu^* = \frac{1}{n}\sum_{i=1}^{n} x_i$ and $\sigma^{2*} = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu^*)^2$. MLE is easily formulated using our representation. For the common case of MLE over i.i.d. data, which has the mathematical formulation $\theta^* = \arg\max_\theta \prod_i f(x_i;\theta)$, we would write:

```
Axiom PROD : forall {n}, R^n -> R.

Definition MLE {T} {A} `{stocked A} {n : nat} (P : T -> dist A)
               `{forall t, has_pdf (P t)} (X : A^n) : option T
:= argmin ( fun t => - PROD (fun i => pdf (P t) (X i)) )
          ( fun t => True ).
```

Aside from the increased level of precision that is required for mechanization, this code is mostly a direct translation of the mathematical statement. The function `MLE` has two

explicit parameters: a family of distributions `P` over a space `A`, indexed by a parameter of type `T` (a mnemonic for "theta"); and a dataset `X` containing `n` observations of type `A`. The remaining parameters are implicit and do not have to be provided by the programmer; these include the proof that `A` is stocked and the proof that the entire family `P` has a PDF. The function returns a value of type `option T`, which is of the form `Some t` or `None` depending on whether or not an optimal parameter value `t` exists. The definition uses a variant of the `min` operator named `argmin`, which returns the argument that achieves the optimal value in a minimization problem, when it exists. The subexpression `pdf (P t)` corresponds directly to the function $f(\cdot\,;\theta)$, which we apply to the i-th data point in both formulations. To express maximization via `argmin`, we negate the original objective function.

4.5.2 Solving MLE with expectation maximization

There does not always exist a simple and straightforward closed-form expression for maximum likelihood estimates. Consider the dataset $x_1, ..., x_n$ of numbers from before, but where we now posit that they are generated by a *mixture* of two Gaussians instead of by a single Gaussian. This new process can be understood as flipping a coin and then drawing a point from one of the Gaussian components based on the result of the coin flip (for simplicity we consider an unbiased coin and unit variance for the normals). This model does not have a convenient likelihood function.

The *expectation maximization* (EM) algorithm is a technique for finding MLE estimates in such cases [17, 39, 16]. More accurately, EM is an algorithm template: different models have different instantiations of the algorithm. EM leverages the fact that, in many cases, models that have inconvenient likelihood functions still have a convenient *joint* likelihood, when considering the *hidden variables* of the model, in addition to the data variables. In our example, the hidden variables would be the the result of each coin flip $z_i \in \{0, 1\}$ that generated each data point x_i. In a sense, the problem is "easy" if only we knew what values the hidden variables have taken. In this vein, the standard interpretation of EM casts it as an iterative algorithm that alternates between hallucinating values for the hidden variables (the *E-step*) and re-optimizing for the best values of the parameters (the *M-step*) based on

these hallucinations. EM is numerically robust and is guaranteed to make progress toward a local maximum.

```
Given x as input:
θ_curr := <initialize_randomly>;
while (<convergence_check>) {
  J(θ) := E_{z|x;θ_curr}[ log f(x, z; θ) ];
  θ_curr := arg max_θ J(θ);
}
return θ_curr;
```

Figure 16: General form of the EM algorithm.

The general form of the EM algorithm is shown in Figure 16. The algorithm takes observed data as input and proceeds by iteratively re-estimating the parameters, starting from a random initialization. The re-estimation step creates a new guess for the parameter from an old guess by selecting the value that maximizes the expected joint log-likelihood. We do this because we do not actually know the values of the hidden variables, so we take the expectation over all possibilities for the hidden variables. This averaging takes into account what we know: the observed data and the current guess for the parameters. The specific instantiation of EM for our running example is shown in Figure 17.

```
Given x as input:
(θ1, θ2) := <initialize_randomly>;
while (<convergence_check>) {
  for i = 1 to n do
    γ_i := φ(x_i − θ1) / ( φ(x_i − θ1) + φ(x_i − θ2) );
  θ1 := ( Σ_{i=1}^n γ_i * x_i ) / Σ_{i=1}^n γ_i ;
  θ2 := ( Σ_{i=1}^n (1 - γ_i) * x_i ) / Σ_{i=1}^n (1 - γ_i);
}
return (θ1, θ2);
```

Figure 17: EM for a mixture of two Gaussians. The function ϕ is the PDF of the normal distribution.

4.5.3 Mechanizing expectation maximization

The most important step in mechanizing EM is to define precisely what is done in the core step of the algorithm. This is implemented as follows:

```
Definition EM_step {T Z X} `{stocked Z} `{stocked X} {n}
                   (P : T -> dist (Z * X)) `{forall t, has_pdf (P t)}
                   (x : X ^ n) (t_prev : T)
           : option T
:= let f t := pdf (P t) in
   let Q t := - INT ( fun z : Z^n =>
                      PROD (fun i => f t_prev (z i, x i) /
                                     INT (fun z : Z => f t_prev (z, x i)))
                      * SUM  (fun i => ln (f t (z i, x i))) ) in
   argmin Q (fun _ => True).
```

This function corresponds to the loop body in Figure 16 and describes how to compute a new estimate of the parameter from a previous estimate `t_prev`. It combines the E-step and M-step. There are three explicit parameters: the previous estimate `t_prev`, the observed data `x`, and the parameterized joint distribution `P` of the hidden and data variables. The other parameters are implicitly determined by the system, including a proof that `P` has a PDF for any setting of its parameter. Integration with the `PROD` term in the definition of `Q` computes exactly the conditional expectation from Figure 16, specialized to the i.i.d. case and written in terms of the joint density. This is done in the standard way, normalizing the joint density by the marginal density of the data, which is itself achieved by integrating out the hidden variable from the joint density. These integrations are performed by the `INT` operator, which corresponds to abstraction integration and reduces to ordinary summation for finite types.

We are now ready to state the necessary theorem. First, we define a new combinator for constructing distributions, named `extend`, in the style of the *extend* operation in the measure transformer semantics of Fun [11]:

```
Definition extend {A B} `{stocked A} `{stocked B}
                  (PA : dist A) (PB : A -> dist B)
           : dist (A * B)
  := var a ~ PA in var b ~ PB a in ret (a,b).
```

This is similar to the `bind` operation and simply creates a hierarchical model. Unlike `bind`, it remembers the first random variable, and returns a joint distribution over *both* of the random variables that are introduced. The EM rewrite is now simply

```
Axiom EM_thm :
  forall {T Z X} `{stocked Z} `{stocked X}
         (P1 : T -> dist Z) (P2 : T -> Z -> dist X)
         `{forall t, has_pdf (bind (P1 t) (P2 t))}
         `{forall t, has_pdf (extend (P1 t) (P2 t))}
         {n} (x : X^n) (t_init : T) (steps : nat),

    MLE (fun t => bind (P1 t) (P2 t)) x
      = EM_loop (fun t => extend (P1 t) (P2 t)) x t_init steps.
```

At a high level this asserts that maximum likelihood problems for hierarchical models of a particular form can be solved by the EM algorithm. In particular, the models must be of the form `bind (P1 t) (P2 t)`, which are models where a random variable drawn from the parameterized distribution `P1` is used to construct a random variable from the data distribution `P2`. The use of `bind` is what makes the draw from `P1` a *hidden* variable. As before, `x` is a dataset of observations. The function `EM_loop` applies `EM_step` to the initial estimate `t_init` for the specified number of iterations, `steps`, to produce a final estimate for the parameter. Note that the argument to `EM_loop` is the parameterized *joint* model, obtained by using `extend` instead of `bind`. The necessary preconditions are computed automatically, such as the requirement that the marginal and joint models each have a PDF.

4.5.4 Remarks on semantic preservation

We are mildly abusing the equality relation `=` provided by the Coq standard library, which raises questions regarding semantic preservation. For our current investigation, this abuse turns out to be harmless.

The EM theorem *as stated* does not actually hold for several reasons. First, EM is only guaranteed to make progress to a *local* optimum and may not equal the global optimum. This is often the case in solvers for nonlinear optimization problems; furthermore, there is not much more that can be said to characterize the quality of the approximation. Second, the equality is stated to hold for any initial estimate and any number of iterations, which is not true. Given this, how would we phrase the semantic preservation theorem for EM if we cannot characterize the nature of the approximation being made? If we cannot, what was the purpose of bothering to define a formal semantics in the first place, if not to support semantic preservation?

Unfortunately, there cannot be a good answer to these questions in the general case: there are many methods used in practice whose approximation guarantees cannot be characterized, and any pragmatic system will need an "escape hatch" to incorporate such techniques. However, this does not mean it was a waste to define a formal semantics. There are numerous styles of approximation algorithms, many of which *can* be characterized. Furthermore, the nature of the guarantee can vary: consider constant factor approximations vs. approximation algorithms with probabilistic guarantees. The task of structuring an algorithm derivation system in the presence of approximation algorithms—as well as defining and mechanizing how the guarantees compose—is an interesting open research question. A formal semantics certainly plays a part in understanding such a system and serves as a basis on which to state semantic preservation theorems (as equalities or as suitable approximation statements) for derivations, or fragments thereof, that do not use the escape hatch.

Concretely, one might define a new relation `f =~ g`, expressing when a function `g` approximates a function `f`, and use this relation in the rewrite theorems ("`EM_loop` approximates `MLE`") and problem statement theorems ("there exists a term that approximates my input specification"). Term rewriting would be achieved via Coq's support for generalized

rewriting with user-defined relations (which we are in fact already using, in conjunction with functional extensionality, in order to rewrite under variable binders). This relation could be implemented as a Coq inductive data type, with a separate case for each kind of approximation algorithm.

In any case, we have chosen to use the equality relation because it is simple and sufficient to investigate the issue of whether machine learning principles can be mechanized. We consistently use equality in a "uni-directional" way—using only left-to-right rewriting—and thus are treating it as an "approximates" relation.

4.5.5 Mechanizing expectation maximization for a mixture of Gaussians

We demonstrate the mechanization of EM by applying it to our running example of a mixture of two Gaussians with unit variance. Again, the corresponding Coq code will be in nearly one-to-one correspondence with the mathematical statement. We start with the definition of a normal distribution parameterized by just its mean

```
Definition normal (t : R) : dist R := var x ~ std_normal in ret (x + t).
```

and use this to write a model for the mixture of two such normals:

```
Definition mo2g (t : R * R) : dist R :=
  var z ~ flip (/2) in (if z then normal (fst t) else normal (snd t)).
```

Note that we parameterize `mo2g` by a single parameter of type `R * R` rather than two parameters of type `R`. This is done to fit into the form expected by `MLE`; the maximum likelihood estimate for this example can now simply be written as `MLE mo2g x`, where `x` is some dataset of observations. Preparing to solve this maximum likelihood problem proceeds as before: we propose the existence of a term that computes the MLE solution:

```
Theorem mo2g_EM : forall {n} (x : R^n) (t_init : R*R) (steps : nat),
                    {ans | ans = MLE mo2g x}.
```

One detail here is that the initial parameter estimate, `t_init`, and the number of iterations, `steps`, are part of the theorem statement, which seems unnecessary: why should those details arise before we have even decided which technique to use to solve this MLE problem?

```
n : nat
x : R ^ n
t_init : R * R
steps : nat
============================
 ?163383 =
 EM_loop
   (fun t : R * R =>
    extend (flip (/ 2))
      (fun z : bool => if z then normal (fst t) else normal (snd t))) x
   t_init steps
```

Figure 18: Proof state immediately after applying the EM reformulation.

This turns out to be necessary because of how we have chosen to emulate algorithm derivation (using equality theorems on output values) and is not a fundamental shortcoming of our syntactic formalization. The key issue is that we need a way to represent the idea that solution techniques can introduce parameters of their own, in addition to the parameters of the original problem. This could be achieved by way of the relation for approximation algorithms suggested in the previous section. We will continue with the basic version, as it is enough to demonstrate that our formalization can mechanize EM.

We examine this derivation by its major steps. The first step simply applies the EM rewrite, after some clerical operations:

```
Proof.
  (* Step 1: Apply the EM rewrite *)
  intros; eapply exist; unfold mo2g; rewrite (EM_thm _ _ _ t_init steps).
```

The major arguments to `EM_thm` are inferred, from unification against the input specification. The proof state after this step is shown in Figure 18; this is exactly the right-hand side of `EM_thm`, specialized to the distributions in our example. The question mark variable represents an existentially quantified variable that has yet to be instantiated and corresponds to the term we are in the process of deriving. The next step unfolds some definitions so we can inspect the details of the EM step, and the resulting proof state is shown in Figure 19:

```
  (* Step 2: Inspect the EM step *)
```

```
n : nat
x : R ^ n
t_init : R * R
steps : nat
============================
 ?163383 =
 match_nat steps (Some t_init)
   (fun acc : option (R * R) =>
    match_option acc None
      (fun t : R * R =>
       argmin
         (fun t0 : R * R =>
          - INT
            (fun z : bool ^ n =>
             PROD
               (fun i : fin n =>
                (if z i then / 2 else 1 - / 2) *
                (if z i then phi (x i - fst t) else phi (x i - snd t)) /
                INT
                  (fun z0 : bool =>
                   (if z0 then / 2 else 1 - / 2) *
                   (if z0 then phi (x i - fst t) else phi (x i - snd t)))) *
             SUM
               (fun i : fin n =>
                ln
                  ((if z i then / 2 else 1 - / 2) *
                   (if z i then phi (x i - fst t0) else phi (x i - snd t0))))))
         (fun _ : R * R => True)))
```

Figure 19: Proof state inspecting the details of the EM step.

```
unfold EM_loop, EM_step; simpl.
```

We see here the guts of `EM_loop` and `EM_step`. The function `match_nat` is the EM "main loop", recursing on the number of iterations: on zero iterations it returns the initial parameter estimate `t_init` and otherwise does the EM step. The function `match_option` handles possibly failing computations: if the optimization in the EM step becomes undefined at some iteration, it returns `None` as the result of the entire computation; otherwise it computes the new parameter estimate from the old one, as defined in `EM_step`. The integration is performing the conditional expectation over the hidden variable; in this case it is expressing an "integration" over `bool^n` (Boolean vectors of length `n`) which is simply a sum over all 2^n possible assignments of the hidden variables. Note that the EM step in the final derived algorithm will only have $O(n)$ complexity.

Next we use the fact that the optimum value will be located where the gradient of the expected joint log-likelihood (function `Q` in `EM_step`) is equal to zero. We accomplish gradient calculation in a similar fashion as PDF calculation, by encoding it as a typeclass. The main definition is

```
Class has_gradient {A} `{euclidean_space A} (f : A -> R) := {
  grad : A -> A
}.
```

which introduces an operator `grad` that takes a real-valued function `f` on a Euclidean space `A` and produces a vector field (of type `A -> A`) corresponding to the gradient. Each component of the gradient vector corresponds to the partial derivative in that direction. We define finite products of the real line `R` as Euclidean spaces; instantiations of `A` include vectors `R^n`, pairs `R * R`, and even more exotic aggregations like `R^n * R * R^k`. We define rules for calculating gradients (*i.e.* `Instance`s of `has_gradient`) that include logic for handling the usual arithmetic operations, in addition to indexed summation `SUM` and tuple projections `fst` and `snd`.

The main command in this step is `grad_opt`, which encodes the gradient condition, applying `grad` to the expected joint log-likelihood; the remaining commands simply perform

```
n : nat
x : R ^ n
t_init : R * R
steps : nat
============================
 ?163383 =
 match_nat steps (Some t_init)
   (fun acc : option (R * R) =>
    match_option acc None
      (fun t : R * R =>
       argmin (fun _ : R * R => 0)
         (fun x0 : R * R =>
          - INT
            (fun z : fin n -> bool =>
             PROD ( ... ) *
             SUM (fun i : fin n => if z i then x i - fst x0 else 0)) = 0 /\
          - INT
            (fun z : fin n -> bool =>
             PROD ( ... ) *
             SUM (fun i : fin n => if z i then 0 else x i - snd x0)) = 0)))
```

Figure 20: Proof state after applying the gradient condition for optimality.

some algebraic simplifications:

```
(* Step 3: Apply the gradient condition *)

unfold phi; clrewrite_strat (bottomup (hints log_rules)).

setoid_rewrite grad_opt; simpl.

clrewrite_strat (bottomup (hints arith_rules)).

setoid_rewrite fst_if; setoid_rewrite snd_if.

setoid_rewrite prod_eq.
```

The proof state after this step is shown in Figure 20. Note that the optimization problem has changed from having a non-trivial objective function with a trivial constraint (`fun _ : R * R => True`) to one with a trivial objective function (`fun _ : R * R => 0`) and a non-trivial constraint. We elide the body of the indexed multiplication `PROD` to keep the presentation simple; it is the same body from the previous step. This new constraint simply states that the gradient must equal zero, *i.e.* that both partial derivatives (w.r.t. each of the two parameters) must equal zero. Note that because the gradient rules use the

name `x` (by convention) for naming the variable of differentiation, Coq has (safely) renamed our parameter `t` to `x0` (the `0` is appended to avoid collision with our name for the dataset, which incidentally is also `x`). The last major step is to solve this system of equations, which happens to be a linear system.

```
    (* Step 4: Solve the resulting system of equations *)
    setoid_rewrite eqn_soln1; setoid_rewrite eqn_soln2.
    setoid_rewrite INT_iid1; setoid_rewrite INT_iid2.
    setoid_rewrite solved_constraint.
    setoid_rewrite bool_INT; setoid_rewrite bool_INT.
    simpl. clrewrite_strat (bottomup (hints arith_rules)).
```

This produces the final proof state shown in Figure 21. This big term actually has a simple form: the EM step in the main EM loop (*i.e.* the body inside of `match_nat` and `match_option`) simply takes the previous estimate `t` and produces a new estimate. This new estimate is a pair where each component is a quotient of two `SUM`s, exactly of the form we originally discussed when we introduced EM, shown in Figure 17. We conclude with

```
    reflexivity.
  Defined.
```

which ends the proof and binds this derived algorithm (together with a proof of its equivalence to our input problem) to the name `mo2g_EM`. We can now use Coq's `Extraction` facility to automatically generate an executable OCaml version of this derived algorithm.

4.6 Lessons learned from using a foundational type theory

We now reflect on some anecdotal lessons we have learned about the benefits of using type theory for our formalization. In particular, we consider lessons from working in the framework of a *foundational type theory* (FTT), such as the Calculus of Constructions or Martin-Löf's intuitionistic type theory. Some of these lessons will be all too familiar to devotees of these type theories, but they deserve to be repeated to the uninitiated.

```
n : nat
x : R ^ n
t_init : R * R
steps : nat
============================
 ?163383 =
 match_nat steps (Some t_init)
   (fun acc : option (R * R) =>
    match_option acc None
      (fun t : R * R =>
       Some
         (SUM
            (fun i : fin n =>
             / 2 * / sqrt (2 * PI) *
             exp (- ((x i - fst t) * (x i - fst t) * / 2)) /
             (/ 2 * / sqrt (2 * PI) *
              exp (- ((x i - fst t) * (x i - fst t) * / 2)) +
              / 2 * / sqrt (2 * PI) *
              exp (- ((x i - snd t) * (x i - snd t) * / 2))) *
             x i) /
          SUM
            (fun i : fin n =>
             / 2 * / sqrt (2 * PI) *
             exp (- ((x i - fst t) * (x i - fst t) * / 2)) /
             (/ 2 * / sqrt (2 * PI) *
              exp (- ((x i - fst t) * (x i - fst t) * / 2)) +
              / 2 * / sqrt (2 * PI) *
              exp (- ((x i - snd t) * (x i - snd t) * / 2)))),
         SUM
           (fun i : fin n =>
            / 2 * / sqrt (2 * PI) *
            exp (- ((x i - snd t) * (x i - snd t) * / 2)) /
            (/ 2 * / sqrt (2 * PI) *
             exp (- ((x i - fst t) * (x i - fst t) * / 2)) +
             / 2 * / sqrt (2 * PI) *
             exp (- ((x i - snd t) * (x i - snd t) * / 2))) *
            x i) /
         SUM
           (fun i : fin n =>
            / 2 * / sqrt (2 * PI) *
            exp (- ((x i - snd t) * (x i - snd t) * / 2)) /
            (/ 2 * / sqrt (2 * PI) *
             exp (- ((x i - fst t) * (x i - fst t) * / 2)) +
             / 2 * / sqrt (2 * PI) *
             exp (- ((x i - snd t) * (x i - snd t) * / 2)))))))
```

Figure 21: Final proof state, depicting the derived algorithm.

Lesson 1: Types guide us when writing highly abstract code. This is a commonly cited benefit of using static type checking, particularly in the presence of *parametric polymorphism* (*i.e.* generics). Consider the definition of `EM_step` or `EM_thm` from Section 4.5.3. The type system enables us to write definitions that abstract over the possible types of the hidden and observed variables (*i.e.* they are not constrained to be particular types).

However, unlike dynamically typed languages, the definitions must still obey the types; specifically, by the property of *parametricity* [66], terms with parametric types can only be combined in certain ways, much like jigsaw puzzle pieces. This helps the programmer by catching programs which appear to put the right expressions in the right places, but are incorrect in subtle ways. This was our experience with `EM_step`, where there is a subtle interplay between the PDF, indexed vs. non-indexed versions of the hidden variables, and the observed data—our first attempts at a definition were type-incorrect. After carefully examining the types, we were guided toward the correct definition. Catching this error statically was especially important for us, because the notion of "run time" is far more delayed in our setting than in general-purpose programming; we have many more steps between the high-level mathematical specification of input problems and the corresponding executable code, so there is a high cost to leaving error-finding to run time.

Lesson 2: FTT gives us a way to be pragmatic about verification of program transformations. There are many approaches to verifying that programs meet certain specifications. An attractive feature of FTT is that terms and meta-level statements about those terms (*e.g.* when they are equal) are written in the same language. Conceptually, this matches what we do on paper, where we use the one language of mathematics. Furthermore, this feature allows us to write so-called "strong specifications" for stating our problem statement theorems. Proving these theorems amounts to chaining together the rewrite theorems and the typeclass instances—which are themselves semantic preservation theorems—to produce a derived algorithm and corresponding proof of correctness. Fortunately, implementations of FTT such as Coq allow us to state these auxiliary theorems axiomatically, without needing to prove everything from first principles all at once. Instead,

we can prove them over time, while in the meantime use them for algorithm derivation. This represents a nice pragmatic approach to building up a database of rewrite theorems, while remaining in a formal and fully verifiable framework.

Lesson 3: FTT greatly clarifies our syntactic understanding of mathematical objects. This is the biggest benefit of FTT, in our opinion. As mentioned before, FTT takes a syntactic approach to laying a foundational system for mathematics, making it suitable for those who care to mechanize mathematics. Furthermore, the core of these theories are composed of only a few primitives. Consequently, any formalization of mathematical objects is constrained to use only these primitives. In particular, variable binding constructs such as `min` and `var` must be implemented using existing variable binders: either lambda abstraction or universal quantification, which are written `(fun x => E)` and `(forall x, E)` in Gallina, respectively (`x` can appear free in the term `E`). Adopting FTT's worldview on how to formalize mathematics turns out to have a significant impact on our ability to understand mathematical objects syntactically.

The process of converting informal mathematics-on-paper to syntactic objects often involves dealing with the low-level issue of managing variable names and variable contexts, and this is the source of many headaches. Compiler writers can attest to the fact that programs with variables are the trickiest kind of data to write algorithms for. Our PDF compiler is such an example; the judgments in Section 2.5.3 take great care to keep track of which variables in the context are random variables and which are parameters. It manipulates open terms and in certain places relies on variable names coinciding in order to produce a PDF. The correctness of this program transformation depends crucially on getting this bookkeeping right, which is in fact extremely tedious and error-prone. By contrast, our implementation of the *exact same logic* in Coq performs no such low-level variable management, eliminating an entire class of variable mis-management errors. This is because Coq provides only a *controlled* interface to syntactic manipulation, and it forces the user to think at the level of proper mathematical objects instead of manipulating open terms, by handling open terms and contexts under the hood.

For instance, compare the typeclass instance `has_pdf_translate` (Section 4.3) against P-LINEAR, its corresponding rule from the PDF compiler. Instead of reasoning about a real-valued open term and carefully keeping track of random variables versus parameters (as P-LINEAR does), `has_pdf_translate` reasons on a real-valued object `(f : A -> R)` that is *explicitly parameterized* by the random variable is depends on. The role of the probabilistic context is played by the distribution `P`, of type `dist A`. Likewise, we see `EM_thm` operating on explicitly parameterized distributions `(P1 : T -> dist Z)` and `(P2 : T -> Z -> dist X)`, rather than operating on a non-parameterized distribution and searching it for hidden variables that are implicit in its structure. Note that the COQ user still writes their programs with the implicit structure; the system unifies against it to determine the different explicit pieces. A potential downside of this controlled interface is that it is much harder to arbitrarily manipulate collections of variables, as was done in a previous hand-rolled prototype for EM [7]. Designing a mechanism that is as safe as the COQ solution but retains (most of) the flexibility of arbitrary variable manipulation remains an interesting research problem.

4.7 Conclusion

We have presented a unified syntactic theory of machine learning. This formalization allows us to express machine learning problems and mechanize useful solution principles in a way that is quite similar to their original mathematical formulations. We have implemented the ideas in the COQ proof assistant using type theory, highlighting the ways in which correctness is promoted in our setup. We hope that this work will consolidate the community's understanding of existing work and spur the next generation of languages for machine learning. In particular, we hope this work not only provides a way to formally understand previous work such as AutoBayes but also encourage future languages to start incorporating primitives beyond probability distributions in their definitions.

CHAPTER V

CONCLUSION

We now review the thesis statement as well as the evidence provided in the dissertation that substantiates the thesis. We also discuss implications of the dissertation and directions for future work raised by this work.

5.1 Review of thesis statement and dissertation

Recall the thesis statement:

> It is possible to construct a syntactic representation of machine learning that is expressive, promotes correctness, and can mechanize useful solution principles.

We substantiate this existence proposition by providing a witness: a syntactic representation of machine learning with the necessary properties.

Expressive. The first aspect of expressivity that we provide is the ability for users to specify both probability and optimization in their programs, in a rich way. Specifically, we provide constructs for probability distributions, probability density functions, and optimization problems. These can be nested and combined arbitrarily, allowing users to represent their learning problems in almost a direct correspondence with their respective mathematical formulations, as we show in several examples.

The second aspect of expressivity involves our use of the probability monad. Structuring our language around the probability monad allows us to express deterministic transformations of random variables, which arises in the ecological models we use in our empirical results. Crucially, it also allows us to implement our language as an embedded domain-specific language in Gallina, the term language of the Coq proof assistant. The embedding allows us to inherit several features of Gallina, such as its compositionality.

Promotes correctness. To promote correctness, we take a formal approach to language design. In particular, we develop our language as an embedded language in Gallina and inherit a well-understood type system and semantics. The consequence is that we have a precise mathematical meaning for any program we write in our language. So, even for programs which arbitrarily mix random variables and optimization variables, we have an unambiguous understanding of what it is specifying. Compare this with AutoBayes, which is not formally defined; it is unclear what such arbitrary combinations actually mean, precisely. We believe this work could serve as a formal foundation for future versions of AutoBayes.

Additionally, with a formal semantics, there is a finally a framework that can be used for reasoning about the behavior of program transformations. This can be used informally, by human users who inspect programs before and after a rewrite. This can also be used formally, by stating and proving semantic preservation theorems. As mentioned before, this also supports a future agenda of *semantic approximation theorems*, to handle the ubiquitous notion of approximation in machine learning.

We demonstrate this claim by implementing our program transformations as theorems in Coq. These are stated axiomatically for pragmatism, but they can be proven from first principles over time to fully verify the program transformations. Compare this with the compilers from related works; though we rely on the writers of the program transformations to encode them correctly—as related works do—we have a framework in which we can migrate toward a fully verified compiler.

Can mechanize useful solution principles. To verify that our theory has real-world benefit, we use it to mechanize several useful techniques, exercising different aspects of the theory.

First, we define a syntactic framework for probability density functions, which lets us write a pdf compiler for producing pdfs from probabilistic programs. We apply this compiler to Bayesian inference problems from ecology by automating the use of Filzbach, an MCMC-based sampler, and we achieve tremendous code savings at modest additional computational cost, which greatly reduces developer burden.

Next, we implement reformulations from optimization. In particular, we formalize the big-M method for disjunctive constraints, based on the formalization of the convex-hull method for disjunctive constraints formalized in Tyles, as defined by Agarwal. We also compare these methods against state-of-the-art solutions, finding that no one method is universally the best option. This underscores the importance of automation, as we do not want the user to manually experiment with these alternative in their quest to find the most efficient reformulation.

Furthermore, we modify this big-M compiler—which deals with discrete choice arising in the *constraint* of an optimization problem—to handle discrete choice that arises in the *objective function* of an optimization problem. Namely, we define a new compiler that handles L_0 regularization terms. Such terms are often added in machine learning problems to induce solution sparsity, and their discrete nature often cannot be handled directly, requiring some sort of reformulation. We show its utility by using it to solve L_0 support vector machines, yielding the recent technique of mixed-integer support vector machines.

Finally, we mechanize the expectation maximization algorithm for solving maximum likelihood estimation problems. Specifically, we mechanize EM for estimating the means of a mixture of two Gaussians. The mechanization depends crucially on the previous two features: PDFs and optimization. Furthermore, it uses the ability to nest optimization problems *inside* of expressions, as the core EM step occurs inside of a loop and is expressed with optimization.

5.2 Directions for future work

While this dissertation lays a firm expressive foundation for the syntactic study of machine learning problems and reformulations, several question remain to be resolved in order to take this work past the state of AutoBayes and to accomplish the fully realized vision of (semi-)automatic derivation of machine learning algorithms. Anecdotal accounts tell us that the AutoBayes codebase of program transformations reached a plateau in size, after which adding new reformulations proved to be quite a burden. We believe there are several reasons for this. First, the AutoBayes language lacks a formal specification (something this

work provides), which can aid in understanding and guiding the development of program transformations.

Perhaps more important is that authoring these transformations in AutoBayes is a relatively crude activity, as it is in most compilers; authors often must deal with low-level variable management logic. This may be manageable in the traditional view of languages and compilers, where the responsibility of writing program transformations lies solely with the compiler writers. However, this does not translate to the vision of such a system as an assistant to a machine learning researcher, who we expect to invent new algorithm templates (such as EM) and encode them in the system. Here, our end-user (the researcher) is interested in writing the programs *and* their transformations, and they will not be well-versed in writing compiler algorithm, nor all the issues and pitfalls that follow.

Another factor pushing us toward end-users being involved with program transformation is the issue of interfacing with existing solvers and solution schemes. The expressivity of our language is good for lowering the semantic gap between thoughts and code, but it is in tension with our ability to actually solve arbitrary problem statements. In reality, a major contribution of our language is to serve as a common representation from which we can compile to many existing solution techniques, each of which operate within different subsets of the full-blown expressive language.

In light of this discussion, we see a few important issues to resolve:

Easing the burden of writing program transformations. As discussed, there will be a shift toward having the end-user write program transformations. An open research question is: how can we make this process less error-prone, particularly in the face of variable management issues? Our Coq implementation is one possible approach; we used the built-in unification capability of the Coq system to pattern match against occurrences of variable binding. Thus, the user never explicitly manages variables; instead, she writes such program transformations as statements on *functions*. To be most effective, this approach needs to be extended to cleanly handle multiple variables that arise in multivariate functions. In general, we wish to work out a mechanism for writing program transformations declaratively.

Interfacing with side-effecting APIs. Interfacing with existing libraries and software is an unavoidable fact. This raises the question of how can we represent this interface, semantically. Our language thus far corresponds to an idealized mathematical concept: all functions are total, and thus do not fail or loop indefinitely. However, real software exhibits both of these properties. One possible solution is to use a "partiality monad", which uses the type system to separate pure computations from possibly side-effecting ones. It remains to be seen how far this approach can be taken.

Generalizing the scope of probabilistic programming. There are many methods which do not strictly fall under the probabilistic worldview, but which are nonetheless widely used. For instance, in supervised learning, a probabilistic programming language wishes to learn a value of type `A -> dist B` because it views supervised learning as conditional density estimation; however, you might also solve the same learning problem with a decision tree that learns a function of type `A -> B`. We believe the idea of probabilistic programming needs to be generalized to include a more general notion of learning, instead of focusing on just the probabilistic account.

REFERENCES

[1] Agarwal, A., *Logical Modeling Frameworks for the Optimization of Discrete-Continuous Systems*. PhD thesis, Carnegie Mellon University, 2006.

[2] Agarwal, A., Bhat, S., Gray, A., and Grossmann, I. E., "Automating mathematical program transformations," in *Practical Aspects of Declarative Languages (PADL)*, 2010.

[3] Audebaud, P. and Paulin-Mohring, C., "Proofs of Randomized Algorithms in Coq," in *Mathematics of Program Construction*, pp. 49–68, Springer, 2006.

[4] Balas, E., "Disjunctive programming: Properties of the convex hull of feasible points," *Discrete Applied Mathematics*, vol. 89, no. 1-3, pp. 3–44, 1998.

[5] Balas, E., "Disjunctive programming: Properties of the convex hull of feasible points," Tech. Rep. MSRR 348, Carnegie Mellon University, 1974.

[6] Barras, B., Boutin, S., Cornes, C., Courant, J., Coscoy, Y., Delahaye, D., de Rauglaudre, D., Filliâtre, J., Giménez, E., Herbelin, H., and others, "The Coq proof assistant reference manual," *INRIA, version*, vol. 6, no. 11.

[7] Bhat, S., Agarwal, A., Gray, A., and Vuduc, R., "Toward Interactive Statistical Modeling," *Procedia Computer Science*, vol. 1, no. 1, pp. 1835–1844, 2010.

[8] Bhat, S., Borgström, J., Gordon, A. D., and Russo, C., "Deriving probability density functions from probabilistic functional programs," in *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 508–522, Springer, 2013.

[9] Bishop, C. and others, *Pattern recognition and machine learning*. Springer New York:, 2006.

[10] Bisschop, J. and Meeraus, A., "On the development of a general algebraic modeling system in a strategic-planning environment," *Mathematical Programming Study*, vol. 20, no. Oct, pp. 1–29, 1982.

[11] Borgström, J., Gordon, A. D., Greenberg, M., Margetson, J., and Gael, J. V., "Measure Transformer Semantics for Bayesian Machine Learning," in *European Symposium on Programming*, pp. 77–96, 2011.

[12] Burges, C. J. C., "A tutorial on support vector machines for pattern recognition," *Data Mining and Knowledge Discovery*, vol. 2, pp. 121–167, 1998.

[13] Colombani, Y. and Heipcke, T., "Mosel: an extensible environment for modeling and programming solutions," in *4th Intl. Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR'02)* (Jussien, N. and Laburthe, F., eds.), (Le Croisic, France), pp. 277–290, 2002.

[14] Curtin, R. R., Cline, J. R., Slagle, N. P., Amidon, M. L., and Gray, A. G., "MLPACK: A Scalable C++ Machine Learning Library," in *BigLearning: Algorithms, Systems, and Tools for Learning at Scale*, 2011.

[15] Daumé III, H., "HBC: Hierarchical Bayes Compiler," 2007.

[16] Dellaert, F., "The expectation maximization algorithm," 2002.

[17] Dempster, A. P., Laird, N. M., and Rubin, D. B., "Maximum likelihood from incomplete data via the em algorithm," *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 1–38, 1977.

[18] Devroye, L., "Non-Uniform Random Variate Generation," 1986.

[19] Erwig, M. and Kollmansberger, S., "Functional Pearls: Probabilistic functional programming in Haskell," *Journal of Functional Programming*, vol. 16, no. 01, pp. 21–34, 2005.

[20] Fischer, B. and Schumann, J., "AutoBayes: A system for generating data analysis programs from statistical models," *Journal of Functional Programming*, vol. 13, no. 03, pp. 483–508, 2003.

[21] Fourer, R., Gay, D. M., and Kernighan, B. W., "A modeling language for mathematical programming," *Management Science*, vol. 36, no. 5, pp. 519–554, 1990.

[22] Giry, M., "A categorical approach to probability theory," *Categorical Aspects of Topology and Analysis*, vol. 915, 1981.

[23] Goodman, N., Mansinghka, V., Roy, D., Bonawitz, K., and Tenenbaum, J., "Church: a language for generative models," in *Uncertainty in Artificial Intelligence*, 2008.

[24] Gray, A. G., Fischer, B., Schumann, J., and Buntine, W., "Automatic Derivation of Statistical Algorithms: The EM Family and Beyond," in *Advances in Neural Information Processing Systems*, 2003.

[25] Guan, W., Gray, A., and Leyffer, S., "Mixed-Integer Support Vector Machine," *2nd NIPS Workshop on Optimization for Machine Learning*, 2009.

[26] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H., "The weka data mining software: an update," *SIGKDD Explor. Newsl.*, vol. 11, pp. 10–18, Nov. 2009.

[27] Hooker, J. N., *Logic-based methods for optimization: combining optimization and constraint satisfaction*. Wiley-Interscience series in discrete mathematics and optimization, John Wiley & Sons, 2000.

[28] Hoyrup, M., Rojas, C., and Weihrauch, K., "The Radon-Nikodym operator is not computable," in *Computability & Complexity in Analysis*, 2011.

[29] Jeroslow, R. G. and Lowe, J. K., "Modeling with integer variables," *Mathematical Programming Study*, vol. 22, no. December, pp. 167–184, 1984.

[30] Kallrath, J., *Modeling languages in mathematical optimization*, vol. 88 of *Applied optimization*. Boston: Kluwer Academic Publishers, 2004.

[31] Kersting, K. and De Raedt, L., "Bayesian Logic Programming: Theory and Tool," in *Introduction to Statistical Relational Learning*, 2007.

[32] Kiselyov, O. and Shan, C., "Embedded probabilistic programming," in *Working conf. on domain specific lang*, Springer, 2009.

[33] Koller, D. and Friedman, N., *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

[34] Kozen, D., "Semantics of Probabilistic Programs," *Journal of Computer and System Sciences*, vol. 22, no. 3, pp. 328–350, 1981.

[35] Liberti, L., *Techniques de Reformulation en Programmation Mathématique*. L'habilitation à diriger des recherches (HDR), Université Paris IX, Lamsade, 2007. Language: English.

[36] Liberti, L., Cafieri, S., and Savourey, D., "The reformulation-optimization software engine," in *Proceedings of the Third international congress conference on Mathematical software*, ICMS'10, (Berlin, Heidelberg), pp. 303–314, Springer-Verlag, 2010.

[37] Lunn, D., Thomas, A., Best, N., and Spiegelhalter, D., "WinBUGS-a Bayesian modelling framework: concepts, structure, and extensibility," *Statistics and Computing*, vol. 10, no. 4, pp. 325–337, 2000.

[38] McInerny, G. J. and Purves, D. W., "Fine-scale environmental variation in species distribution modelling: regression dilution, latent variables and neighbourly advice," *Methods in Ecology and Evolution*, vol. 2, no. 3, pp. 248–257, 2011.

[39] McLachlan, G. J. and Krishnan, T., *The EM algorithm and extensions*, vol. 382. Wiley-Interscience, 2007.

[40] Mhamdi, T., Hasan, O., and Tahar, S., "On the Formalization of the Lebesgue Integration Theory in HOL," *Interactive Theorem Proving*, pp. 387–402, 2010.

[41] Milch, B., Marthi, B., Russell, S., Sontag, D., Ong, D., and Kolobov, A., "BLOG: Probabilistic models with unknown objects," in *International joint conference on artificial intelligence*, vol. 19, 2005.

[42] Minka, T., Winn, J., Guiver, J., and Kannan, A., "Infer.NET 2.3," 2009. Microsoft Research Cambridge. http://research.microsoft.com/infernet.

[43] Murphy, K. P., "The bayes net toolbox for matlab," *Computing Science and Statistics*, vol. 33, p. 2001, 2001.

[44] Nanevski, A., Blelloch, G., and Harper, R., "Automatic generation of staged geometric predicates," in *Proceedings of the sixth ACM SIGPLAN International Conference on Functional programming, ICFP 2001*, pp. 217–228, Florence, Italy: ACM, 2001.

[45] Neal, R. M., "Probabilistic inference using Markov chain Monte Carlo methods," Tech. Rep. CRG-TR-93-1, Dept. of Computer Science, University of Toronto, September 1993. 144pp.

[46] Nemhauser, G. L. and Wolsey, L. A., *Integer and combinatorial optimization*. Wiley-Interscience series in discrete mathematics and optimization, NY: Wiley, 1999.

[47] Nielsen, O., *An Introduction to Integration and Measure Theory*. Wiley-Interscience, 1997.

[48] Park, S., Pfenning, F., and Thrun, S., "A probabilistic language based upon sampling functions," in *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 171–182, ACM New York, NY, USA, 2005.

[49] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E., "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[50] Pfeffer, A., "IBAL: A probabilistic rational programming language," in *International Joint Conference on Artificial Intelligence*, vol. 17, pp. 733–740, 2001.

[51] Potts, P., Edalat, A., and Escardo, M., "Semantics of exact real arithmetic," in *LICS '97., 12th Annual IEEE Symp. on Logic in Comp. Sci.*, (Warsaw), pp. 248–257, 1997.

[52] Purves, D. and Lyutsarev, V., *Filzbach User Guide*, 2012.

[53] R Core Team, *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2012. ISBN 3-900051-07-0.

[54] Raman, R. and Grossmann, I. E., "Modelling and computational techniques for logic based integer programming," *Computers & Chem. Eng.*, vol. 18, no. 7, pp. 563–578, 1994.

[55] Ramsey, N. and Pfeffer, A., "Stochastic lambda calculus and monads of probability distributions," vol. 37, pp. 154–165, ACM, 2002.

[56] Richardson, M. and Domingos, P., "Markov logic networks," *Machine Learning*, vol. 62, no. 1, pp. 107–136, 2006.

[57] Sato, T. and Kameya, Y., "PRISM: A symbolic-statistical modeling language," in *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97)*, pp. 1330–1339, 1997.

[58] Sawaya, N., *Reformulations, Relaxations and Cutting Planes for Generalized Disjunctive Programming*. PhD thesis, Carnegie Mellon University, 2006.

[59] Scott, D., "Parametric Statistical Modeling by Minimum Integrated Square Error," *Technometrics*, vol. 43, no. 3, pp. 274–285, 2001.

[60] Silverman, B., *Density Estimation for Statistics and Data Analysis*. Chapman & Hall/CRC, 1986.

[61] Smith, M. J., Vanderwel, M. C., Lyutsarev, V., Emmott, S., and Purves, D. W., "The climate dependence of the terrestrial carbon cycle; including parameter and structural uncertainties," *Biogeosciences Discussions*, vol. 9, no. 10, pp. 13439–13496, 2012.

[62] Solovay, R., "A Model of Set-Theory in Which Every Set of Reals is Lebesgue Measurable," *Annals of Mathematics*, pp. 1–56, 1970.

[63] Syme, D., Granicz, A., and Cisternino, A., *Expert F#*. Apress, 2007.

[64] van Hentenryck, P., *The OPL optimization programming language*. Cambridge, Mass.: MIT Press, 1999.

[65] Vecchietti, A. and Grossmann, I. E., "Modeling issues and implementation of language for disjunctive programming," *Computers & Chem. Eng.*, vol. 24, no. 9–10, pp. 2143–2155, 2000.

[66] Wadler, P., "Theorems for free!," in *FUNCTIONAL PROGRAMMING LANGUAGES AND COMPUTER ARCHITECTURE*, pp. 347–359, ACM Press, 1989.

[67] Wasserman, L., *All of statistics: a concise course in statistical inference*. Springer Verlag, 2004.

[68] Wingate, D., Stuhlmueller, A., and Goodman, N. D., "Lightweight implementations of probabilistic programming languages via transformational compilation," in *Proceedings of the 14th international conference on Artificial Intelligence and Statistics*, p. 131, 2011.