Measure Transformer Semantics for Bayesian Machine Learning

Johannes Borgström Andrew D. Gordon
Michael Greenberg James Margetson Jurgen Van Gael

July 2011

Technical Report
MSR-TR-2011-18

Microsoft Research
Roger Needham Building
7 J.J. Thomson Avenue
Cambridge, CB3 0FB
United Kingdom

Publication History

An abridged version of this report appears in the proceedings of the 20th European Symposium on Programming (ESOP'11), part of ETAPS 2011, held in Saarbrücken, Germany, March 26–April 3, 2011.

Measure Transformer Semantics for Bayesian Machine Learning

Johannes Borgström[1], Andrew D. Gordon[1], Michael Greenberg[2], James Margetson[1], and Jurgen Van Gael[1]

[1] Microsoft Research
[2] University of Pennsylvania

Abstract. The Bayesian approach to machine learning amounts to inferring posterior distributions of random variables from a probabilistic model of how the variables are related (that is, a prior distribution) and a set of observations of variables. There is a trend in machine learning towards expressing Bayesian models as probabilistic programs. As a foundation for this kind of programming, we propose a core functional calculus with primitives for sampling prior distributions and observing variables. We define combinators for measure transformers, based on theorems in measure theory, and use these to give a rigorous semantics to our core calculus. The original features of our semantics include its support for discrete, continuous, and hybrid measures, and, in particular, for observations of zero-probability events. We compile our core language to a small imperative language that in addition to the measure transformer semantics also has a straightforward semantics via factor graphs, data structures that enable many efficient inference algorithms. We use an existing inference engine for efficient approximate inference of posterior marginal distributions, treating thousands of observations per second for large instances of realistic models.

1 Introduction

In the past 15 years, statistical machine learning has unified many seemingly unrelated methods through the Bayesian paradigm. With a solid understanding of the theoretical foundations, advances in algorithms for inference, and numerous applications, the Bayesian paradigm is now the state of the art for learning from data. The theme of this paper is the idea of writing Bayesian models as probabilistic programs, which was pioneered by Koller et al. [20] and is recently gaining in popularity [36,35,10,5,18]. In particular, we draw inspiration from Csoft [44], an imperative language with an informal probabilistic semantics. Csoft is the native language of Infer.NET [30], a software library for Bayesian reasoning. A compiler turns Csoft programs into factor graphs [22], data structures that support efficient inference algorithms [19]. This paper borrows ideas from Csoft and extends them, placing the semantics on a firm footing.

Bayesian Models as Probabilistic Expressions Consider a simplified form of TrueSkill [13], a large-scale online system for ranking computer gamers. There is a population of players, each assumed to have a skill, which is a real number that cannot be directly observed. We observe skills only indirectly via a series of matches. The problem is to infer

the skills of players given the outcomes of the matches. Here is a concrete example: *Alice, Bob, and Cyd are new players. In a tournament of three games, Alice beats Bob, Bob beats Cyd, and Alice beats Cyd. What are their skills?* In a Bayesian setting, we represent our uncertain knowledge of the skills as continuous probability distributions. The following probabilistic expression models the situation by generating probability distributions for the players' skills, given three played games (observations).

```
// prior distributions, the hypothesis
let skill() = random (Gaussian(10.0,20.0))
let Alice,Bob,Cyd = skill(),skill(),skill()
// observe the evidence
let performance player = random (Gaussian(player,1.0))
observe (performance Alice > performance Bob) //Alice beats Bob
observe (performance Bob > performance Cyd) //Bob beats Cyd
observe (performance Alice > performance Cyd) //Alice beats Cyd
// return the skills
Alice,Bob,Cyd
```

A run of this expression goes as follows. We sample the skills of the three players from the *prior distribution* **Gaussian**$(10.0, 20.0)$. Such a distribution can be pictured as a bell curve centred on the *mean* 10.0, and gradually tailing off at a rate given by the *variance*, here 20.0. Sampling from such a distribution is a randomized operation that returns a real number, most likely close to the mean. For each match, the run continues by sampling an individual performance for each of the two players. Each performance is centred on the skill of a player, with low variance, making the performance closely correlated with but not identical to the skill. We then observe that the winner's performance is greater than the loser's. An *observation* **observe** *M* always returns (), but represents a constraint that *M* must hold. A whole run is valid if all encountered observations are true. The run terminates by returning the three skills.

A classic computational method to learn the posterior distribution of each of the skills is Monte Carlo sampling [25]. We run the expression many times, but keep just the valid runs—the ones where the sampled skills correspond to the observed outcomes. We then compute the means of the resulting skills by applying standard statistical formulas. In the example above, the *posterior distribution* of the returned skills has moved so that the mean of Alice's skill is greater than Bob's, which is greater than Cyd's. To the best of our knowledge, all prior inference techniques for probabilistic languages with continuous distributions, apart from Csoft and recent versions of IBAL [37], are based on nondeterministic inference using some form of Monte Carlo sampling.

Inference algorithms based on factor graphs [22,19] are an efficient alternative to Monte Carlo sampling. Factor graphs, used in Csoft, allow deterministic but approximative inference algorithms, which are known to be significantly more efficient than sampling methods, where applicable.

Observations with zero probability arise naturally in Bayesian models. For example, in the model above, a drawn game would be modelled as the performance of two players being observed to be equal. Since the performances are randomly drawn from a continuous distribution, the probability of them actually being equal is zero, so we would not expect to see *any* valid runs in a Monte Carlo simulation. (To use Monte Carlo methods,

one must instead write that the absolute difference between two drawn performances is less than some small ε.) However, our semantics based on measure theory makes sense of such observations. Our semantics is the first for languages with continuous or hybrid distributions, such as Fun or Imp, that are implemented by deterministic inference via factor graphs.

Plan of the Paper We propose Fun:

- Fun is a functional language for Bayesian models with primitives for probabilistic sampling and observations (Section 2).
- Fun has a rigorous probabilistic semantics as measure transformers (Section 3).
- Fun has an efficient implementation: our system compiles Fun to Imp (Section 4), a subset of Csoft, and then relies on Infer.NET (Section 6).
- Fun supports array types and array comprehensions in order to express Bayesian models over large datasets (Section 5).

Our main contribution is a framework for finite measure transformer semantics, which supports discrete measures, continuous measures, and mixtures of the two, and also supports observations of zero probability events.

As a substantial application, we supply measure transformer semantics for Fun, Imp, and factor graphs, and use the semantics to verify the translations in our compiler. Theorem 1 establishes agreement with the discrete semantics of Section 2 for Bernoulli Fun. Theorem 2 and Theorem 3 establish the correctness of the first step, from Fun to Imp, and the second step, from Imp to factor graphs.

We designed Fun to be a subset of the F# dialect of ML [43], for implementation convenience: F# reflection allows easy access to the abstract syntax of a program. All the examples in the paper have been executed with our system, described in Section 6. We end the paper with a description of related work (Section 7) and some concluding remarks (Section 8).

Appendix A contains proofs omitted from the main body of the paper. Appendix B lists the code of an F# implementation of measure transformers in the discrete case.

2 Bayesian Models as Probabilistic Expressions

We introduce the idea of expressing a probabilistic model as code in a functional language, Fun, with primitives for generating and observing random variables. For a subset, Bernoulli Fun, limited to weighted Boolean choices, we describe in elementary terms an operational semantics that allows us to compute the conditional probability that the expression yields a given value given that the run was valid.

2.1 Syntax, Informal Semantics, and Bayesian Reading

Expressions are strongly typed, with types t built up from base scalar types b and pair types. We let c range over constant data of scalar type, n over integers, and r over real numbers. We write $\mathrm{ty}(c) = t$ to mean that constant c has type t. For each base type b, we define a *zero element* 0_b. We have arithmetic and Boolean operations on base types.

Types, Constant Data, and Zero Elements:

$a, b ::= \mathbf{bool} \mid \mathbf{int} \mid \mathbf{real}$	base type
$t ::= \mathbf{unit} \mid b \mid (t_1 * t_2)$	compound type
$\mathrm{ty}(()) = \mathbf{unit}$ $\mathrm{ty}(\mathbf{true}) = \mathrm{ty}(\mathbf{false}) = \mathbf{bool}$ $\mathrm{ty}(n) = \mathbf{int}$ $\mathrm{ty}(r) = \mathbf{real}$	
$0_{\mathbf{bool}} = \mathbf{true}$ $0_{\mathbf{int}} = 0$ $0_{\mathbf{real}} = 0.0$	

Signatures of Arithmetic and Logical Operators: $\otimes : b_1, b_2 \to b_3$

$\&\&, \|\|, = : \mathbf{bool}, \mathbf{bool} \to \mathbf{bool}$	$>, = : \mathbf{int}, \mathbf{int} \to \mathbf{bool}$	
$+, -, *, \% : \mathbf{int}, \mathbf{int} \to \mathbf{int}$	$> : \mathbf{real}, \mathbf{real} \to \mathbf{bool}$	$+, -, * : \mathbf{real}, \mathbf{real} \to \mathbf{real}$

We have several standard probability distributions as primitive: $D : t \to u$ takes parameters in t and yields a random value in u. The names x_i below only document the meaning of the parameters.

Signatures of Distributions: $D : (x_1 : b_1 * \cdots * x_n : b_n) \to b$

$\mathbf{Bernoulli} : (\mathsf{success} : \mathbf{real}) \to \mathbf{bool}$
$\mathbf{Binomial} : (\mathsf{trials} : \mathbf{int} * \mathsf{success} : \mathbf{real}) \to \mathbf{int}$
$\mathbf{Poisson} : (\mathsf{rate} : \mathbf{real}) \to \mathbf{int}$
$\mathbf{DiscreteUniform} : (\mathsf{max} : \mathbf{int}) \to \mathbf{int}$
$\mathbf{Gaussian} : (\mathsf{mean} : \mathbf{real} * \mathsf{variance} : \mathbf{real}) \to \mathbf{real}$
$\mathbf{Beta} : (\mathsf{a} : \mathbf{real} * \mathsf{b} : \mathbf{real}) \to \mathbf{real}$
$\mathbf{Gamma} : (\mathsf{shape} : \mathbf{real} * \mathsf{scale} : \mathbf{real}) \to \mathbf{real}$

The expressions and values of Fun are below. Expressions are in a limited syntax akin to A-normal form, with let-expressions for sequential composition.

Fun: Values and Expressions

$V ::= x \mid c \mid (V, V)$	value
$M, N ::=$	expression
$\quad V$	$\quad$ value
$\quad V_1 \otimes V_2$	$\quad$ arithmetic or logical operator
$\quad V.1$	$\quad$ left projection from pair
$\quad V.2$	$\quad$ right projection from pair
$\quad \mathbf{if}\ V\ \mathbf{then}\ M_1\ \mathbf{else}\ M_2$	$\quad$ conditional
$\quad \mathbf{let}\ x = M\ \mathbf{in}\ N$	$\quad$ let (scope of x is N)
$\quad \mathbf{random}\ (D(V))$	$\quad$ primitive distribution
$\quad \mathbf{observe}\ V$	$\quad$ observation

In the discrete case, Fun has a standard *sampling semantics*; the formal semantics for the general case comes later. A run of a closed expression M is the process of evaluating M to a value. The evaluation of most expressions is standard, apart from sampling and observation.

To run $\mathbf{random}\ (D(V))$, where $V = (c_1, \ldots, c_n)$, choose a value c at random, with probability given by the distribution $D(c_1, \ldots, c_n)$, and return c.

To run **observe** V, always return (). We say the observation is *valid* if and only if the value V is some zero element 0_b.

Due to the presence of sampling, different runs of the same expression may yield more than one value, with differing probabilities. Let a run be *valid* so long as every encountered observation is valid. The sampling semantics of an expression is the conditional probability of returning a particular value, given a valid run. Intuitively, Boolean observations are akin to assume statements in assertion-based program specifications, where runs of a program are ignored if an assumed formula is false.

Example: Two Coins, Not Both Tails

```
let heads1 = random (Bernoulli(0.5)) in
let heads2 = random (Bernoulli(0.5)) in
let u = observe (heads1 || heads2) in
(heads1,heads2)
```

The subexpression **random** (**Bernoulli**(0.5)) generates **true** or **false** with equal likelihood. The whole expression has four distinct runs, each with probability $1/4$, corresponding to the possible combinations of Booleans heads1 and heads2. All these runs are valid, apart from the one where heads1 = **false** and heads2 = **false** (representing two tails), since **observe**(**false**||**false**) is not a valid observation. The sampling semantics of this expression is a probability distribution assigning probability $1/3$ to the values (**true**, **false**), (**false**, **true**), and (**true**, **true**), but probability 0 to the value (**false**, **false**).

The sampling semantics allows us to interpret an expression as a Bayesian model. We interpret the distribution of possible return values as the *prior probability* of the model. The constraints on valid runs induced by observations represent new evidence or training data. The conditional probability of a value given a valid run is the *posterior probability*: an adjustment of the prior probability given the evidence or training data.

Thus, the expression above can be read as a Bayesian model of the problem: *I toss two coins. I observe that not both are tails. What is the probability of each outcome?* The uniform distribution of two Booleans represents our prior knowledge about two coins, the **observe** expression represents the evidence that not both are tails, and the overall sampling semantics is the posterior probability of two coins given this evidence.

Next, we define syntactic conventions and a type system for Fun, define a formal semantics for the discrete subset of Fun, and describe further examples. Our discrete semantics is a warm up before Section 3. There we deploy measure theory to give a semantics to our full language, which allows both discrete and continuous prior distributions.

2.2 Syntactic Conventions and Monomorphic Typing Rules

We recite our standard syntactic conventions and typing rules.

We identify phrases of syntax (such as values and expressions) up to consistent renaming of bound variables(such as x in a let-expression). Let $\mathrm{fv}(\phi)$ be the set of variables occurring free in phrase ϕ. Let $\phi\,\{\psi/x\}$ be the outcome of substituting phrase ψ for each free occurrence of variable x in phrase ϕ. To keep our core calculus small, we treat

function definitions as macros with call-by-value semantics. In particular, in examples, we write first-order non-recursive function definitions in the form **let** $f\ x_1\ \ldots\ x_n = M$, and we allow function applications $f\ M_1\ \ldots\ M_n$ as expressions. We consider such a function application as being a shorthand for the expression **let** $x_1 = M_1$ **in** $\ldots$ **let** $x_n = M_n$ **in** M, where the bound variables $x_1, \ldots, x_n$ do not occur free in $M_1, \ldots, M_n$. We allow expressions to be used in place of values, via insertion of suitable let-expressions. For example, (M_1, M_2) stands for **let** $x_1 = M_1$ **in let** $x_2 = M_2$ **in** (x_1, x_2), and $M_1 \otimes M_2$ stands for **let** $x_1 = M_1$ **in let** $x_2 = M_2$ **in** $x_1 \otimes x_2$, when either M_1 or M_2 or both is not a value. Let $M_1; M_2$ stand for **let** $x = M_1$ **in** M_2 where $x \notin \mathrm{fv}(M_2)$. The notation $t = t_1 * \cdots * t_n$ for tuple types means the following: when $n = 0$, $t =$ **unit**; when $n = 1$, $t = t_1$; and when $n > 1$, $t = t_1 * (t_2 * \cdots * t_n)$. In listings, we rely on syntactic abbreviations available in F#, such as layout conventions (to suppress **in** keywords) and writing tuples as $M_1, \ldots, M_n$ without enclosing parentheses.

Let a *typing environment*, Γ, be a list of the form $\varepsilon, x_1 : t_1, \ldots, x_n : t_n$; we say Γ is *well-formed* and write $\Gamma \vdash \diamond$ to mean that the variables x_i are pairwise distinct. Let $\mathrm{dom}(\Gamma) = \{x_1, \ldots, x_n\}$ if $\Gamma = \varepsilon, x_1 : t_1, \ldots, x_n : t_n$. We sometimes use the notation $\overline{x : t}$ for $\Gamma = \varepsilon, x_1 : t_1, \ldots, x_n : t_n$ where $\overline{x} = x_1, \ldots, x_n$ and $\overline{t} = t_1, \ldots, t_n$.

Typing Rules for Fun Expressions: $\Gamma \vdash M : t$

(FUN VAR)
$$\frac{\Gamma \vdash \diamond \quad (x : t) \in \Gamma}{\Gamma \vdash x : t}$$

(FUN CONST)
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash c : \mathrm{ty}(c)}$$

(FUN PAIR)
$$\frac{\Gamma \vdash V_1 : t_1 \quad \Gamma \vdash V_2 : t_2}{\Gamma \vdash (V_1, V_2) : t_1 * t_2}$$

(FUN OPERATOR)
$$\frac{\otimes : b_1, b_2 \to b_3 \quad \Gamma \vdash V_1 : b_1 \quad \Gamma \vdash V_2 : b_2}{\Gamma \vdash V_1 \otimes V_2 : b_3}$$

(FUN PROJ1)
$$\frac{\Gamma \vdash V : t_1 * t_2}{\Gamma \vdash V.1 : t_1}$$

(FUN PROJ2)
$$\frac{\Gamma \vdash V : t_1 * t_2}{\Gamma \vdash V.2 : t_2}$$

(FUN IF)
$$\frac{\Gamma \vdash V : \textbf{bool} \quad \Gamma \vdash M_1 : t \quad \Gamma \vdash M_2 : t}{\Gamma \vdash \textbf{if } V \textbf{ then } M_1 \textbf{ else } M_2 : t}$$

(FUN LET)
$$\frac{\Gamma \vdash M_1 : t_1 \quad \Gamma, x : t_1 \vdash M_2 : t_2}{\Gamma \vdash \textbf{let } x = M_1 \textbf{ in } M_2 : t_2}$$

(FUN RANDOM)
$$\frac{D : (x_1 : b_1 * \cdots * x_n : b_n) \to b \quad \Gamma \vdash V : (b_1 * \cdots * b_n)}{\Gamma \vdash \textbf{random } (D(V)) : b}$$

(FUN OBSERVE)
$$\frac{\Gamma \vdash V : b}{\Gamma \vdash \textbf{observe } V : \textbf{unit}}$$

Lemma 1. *If* $\Gamma, x : t, \Gamma' \vdash M : t'$ *and* $\Gamma \vdash V : t$ *then* $\Gamma, \Gamma' \vdash E\{^V/_x\} : t'$.

Proof. By induction on the derivation of $\Gamma, x : t, \Gamma' \vdash M : t'$. □

Lemma 2. *If* $\Gamma \vdash M : t$ *then* $\Gamma \vdash \diamond$.

Proof. By induction on the derivation of $\Gamma \vdash M : T$. □

Lemma 3 (Unique Types). *If* $\Gamma \vdash M : t$ *and* $\Gamma \vdash M : t'$ *then* $t = t'$.

Proof. By induction on the structure of M. The proof needs that the result types of the signatures of overloaded binary operators and of distributions are determined by the argument types. □

2.3 Formal Semantics for Bernoulli Fun

Let Bernoulli Fun be the fragment of our calculus where every **random** expression takes the form **random** (**Bernoulli**(c)) for some real $c \in (0,1)$, that is, a weighted Boolean choice returning **true** with probability c, and **false** with probability $1-c$. We show that closed well-typed expressions induce conditional probabilities $\mathrm{P}[\mathsf{value} = V \mid \mathsf{valid}]$, the probability that the value of a valid run of E is V.

For this calculus, we inductively define an operational semantics, $M \to^p M'$, meaning that expression M takes a step to M' with probability p.

Reduction Relation: $M \to^p M'$ **where** $p \in (0,1]$

$V_1 \otimes V_2 \to^1 c$ if $V_1 = c_1$, $V_2 = c_2$, and $c = c_1 \otimes c_2$
$(V_1, V_2).1 \to^1 V_1$
$(V_1, V_2).2 \to^1 V_2$
if true then M_1 **else** $M_2 \to^1 M_1$
if false then M_1 **else** $M_2 \to^1 M_2$
let $x = V$ **in** $M \to^1 M\{^V/_x\}$

$\mathcal{R}[M] \to^p \mathcal{R}[M']$ if $M \to^p M'$ for reduction context $\mathcal{R}$ given by
$\mathcal{R} ::= [] \mid$ **let** $x = \mathcal{R}$ **in** M

random (**Bernoulli**(c)) $\to^c$ **true**	if $c \in (0,1)$
random (**Bernoulli**(c)) $\to^{1-c}$ **false**	if $c \in (0,1)$
observe $V \to^1 ()$	

Since there is no recursion or unbounded iteration in Bernoulli Fun, there are no non-terminating reduction sequences $M_1 \to^{p_1} \ldots M_n \to^{p_n} \ldots$.

Moreover, we can prove standard preservation and progress lemmas.

Lemma 4 (Preservation). *If $\Gamma \vdash M : t$ and $M \to^p M'$ then $\Gamma \vdash M' : t$.*

Proof. By induction on the derivation of $M \to^p M'$. □

Lemma 5 (Progress). *If $\varepsilon \vdash M : t$ and M is not a value then there are p and M' such that $M \to^p M'$.*

Proof. By induction on the structure of M. □

Lemma 6 (Determinism). *If $M \to^p M'$ and $M \to^{p'} M'$ then $p = p'$.*

Proof. By induction on the structure of M. □

Lemma 7 (Probability). *If $\varepsilon \vdash M : t$ then $\Sigma_{\{(p,N) \mid M \to^p N\}} p = 1$.*

Proof. By induction on the structure of M. □

We consider a fixed expression M such that $\varepsilon \vdash M : t$.

Let the space Ω be the set of all runs of M, where a *run* is a sequence $\omega = (M_1, \ldots, M_{n+1})$ for $n \geq 0$ and $p_1, \ldots, p_n$ such that $M = M_1 \to^{p_1} \cdots \to^{p_n} M_{n+1} = V$;

we define the functions $\mathsf{value}(\omega) = V$ and $\mathsf{prob}(\omega) = 1p_1 \ldots p_n$, and we define the predicate $\mathsf{valid}(\omega)$ to hold if and only if whenever $M_j = \mathcal{R}[\textbf{observe } V]$ then $V = 0_b$ for some zero element 0_b. Since M is well-typed, is normalizing, and samples only from Bernoulli distributions, Ω is finite.

Let an *event*, α or β, be a subset of Ω. Let α and β range over events, and let probability $\mathrm{P}[\alpha] = \sum_{\omega \in \alpha} \mathsf{prob}(\omega)$.

Proposition 1. *The function* $\mathrm{P}[\alpha]$ *forms a* probability distribution*, that is, (1) we have* $\mathrm{P}[\alpha] \geq 0$ *for all* α*, (2)* $\mathrm{P}[\Omega] = 1$*, and (3)* $\mathrm{P}[\alpha \cup \beta] = \mathrm{P}[\alpha] + \mathrm{P}[\beta]$ *if* $\alpha \cap \beta = \varnothing$*.*

Proof. Item (1) follows from the fact that $p \geq 0$ whenever $M \to_p N$. Item (2) follows from Lemma 7, Lemma 4, and termination. Item (3) is immediate from the definition. □

To give the semantics of our expression M we first define the following probabilities and events. Given a value V, $\mathsf{value} = V$ is the event $\mathsf{value}^{-1}(V) = \{\omega \mid \mathsf{value}(\omega) = V\}$. Hence, $\mathrm{P}[\mathsf{value} = V]$ is the odds (or *prior probability*) that a run terminates with V. We let the event $\mathsf{valid} = \{\omega \in \Omega \mid \mathsf{valid}(\omega)\}$; hence, $\mathrm{P}[\mathsf{valid}]$ is the probability that a run is valid.

If $\mathrm{P}[\beta] \neq 0$, the *conditional probability of* α *given* β is

$$\mathrm{P}[\alpha \mid \beta] \triangleq \frac{\mathrm{P}[\alpha \cap \beta]}{\mathrm{P}[\beta]}$$

The semantics of a program is given by the conditional probability distribution

$$\mathrm{P}[\mathsf{value} = V \mid \mathsf{valid}] = \frac{\mathrm{P}\left[(\mathsf{value}^{-1}(V)) \cap \mathsf{valid}\right]}{\mathrm{P}[\mathsf{valid}]},$$

the conditional probability that a run returns V given a valid run, also known as the *posterior probability*.

The conditional probability $\mathrm{P}[\mathsf{value} = V \mid \mathsf{valid}]$ is only defined when $\mathrm{P}[\mathsf{valid}]$ is not zero. For pathological choices of M such as **observe false** or **let** $x = 3$ **in observe** x there are no valid runs, so $\mathrm{P}[\mathsf{valid}] = 0$, and $\mathrm{P}[\mathsf{value} = V \mid \mathsf{valid}]$ is undefined. (This is an occasional problem in practice; Bayesian inference engines such as Infer.NET fail in this situation with a zero-probability exception.)

2.4 An Example in Bernoulli Fun

The expression below encodes the question: *1% of a population have a disease. 80% of subjects with the disease test positive, and 9.6% without the disease also test positive. If a subject is positive, what are the odds they have the disease?* [46]

Epidemiology: Odds of Disease Given Positive Test

```
let has_disease = random (Bernoulli(0.01))
let positive_result = if has_disease
                      then random (Bernoulli(0.8))
                      else random (Bernoulli(0.096))
observe positive_result
has_disease
```

For this expression, we have $\Omega = \{\omega_{tt}, \omega_{tf}, \omega_{ft}, \omega_{ff}\}$ where each run $\omega_{c_1 c_2}$ corresponds to the choice $\mathsf{has\_disease} = c_1$ and $\mathsf{positive\_result} = c_2$. The probability of each run is:

- $\mathsf{prob}(\omega_{tt}) = 0.01 \times 0.8 = 0.008$ (true positive)
- $\mathsf{prob}(\omega_{tf}) = 0.01 \times 0.2 = 0.002$ (false negative)
- $\mathsf{prob}(\omega_{ft}) = 0.99 \times 0.096 = 0.09504$ (false positive)
- $\mathsf{prob}(\omega_{ff}) = 0.99 \times 0.904 = 0.89496$ (true negative)

The semantics $\mathrm{P}\,[\mathsf{value} = \mathbf{true} \mid \mathsf{valid}]$ here is the conditional probability of having the disease, given that the test is positive.

Here $\mathrm{P}\,[\mathsf{valid}] = \mathsf{prob}(\omega_{ft}) + \mathsf{prob}(\omega_{tt})$ and $\mathrm{P}\,[\mathsf{value} = \mathbf{true} \cap \mathsf{valid}] = \mathsf{prob}(\omega_{tt})$, so we have $\mathrm{P}\,[\mathsf{value} = \mathbf{true} \mid \mathsf{valid}] = 0.008/(0.008 + 0.09504) = 0.07764$. So the likelihood of disease given a positive test is just 7.8%, less than one might think.

This example illustrates inference on an explicit enumeration of the runs in Ω. The size of Ω is exponential in the number of **random** expressions, so although illustrative, this style of inference does not scale up. As we explain in Section 4, our implementation strategy is to translate Fun expression to factor graphs, for efficient approximate inference.

Bayes' rule Since our semantics is a conditional probability, Bayes' rule, stated below, provides an alternative procedure to compute it.

$$\mathrm{P}\,[\alpha \mid \beta] = \frac{\mathrm{P}\,[\beta \mid \alpha]\,\mathrm{P}\,[\alpha]}{\mathrm{P}\,[\beta]}$$

We define $r/0 \triangleq 0$. Specialized to our semantics we get that

$$\mathrm{P}\,[\mathsf{value} = V \mid \mathsf{valid}] = \frac{\mathrm{P}\,[\mathsf{valid} \mid \mathsf{value} = V]\,\mathrm{P}\,[\mathsf{value} = V]}{\mathrm{P}\,[\mathsf{valid}]}.$$

The quantities in the rule are usually described as follows:

- $\mathrm{P}\,[\mathsf{value} = V]$ is the *prior probability* that $\mathsf{value} = V$;
- $\mathrm{P}\,[\mathsf{valid} \mid \mathsf{value} = V]$ is known as the *likelihood*, the probability of a valid run given that $\mathsf{value} = V$;
- $\mathrm{P}\,[\mathsf{valid}]$ is the probability of a valid run;
- $\mathrm{P}\,[\mathsf{value} = V \mid \mathsf{valid}]$ is the *posterior probability* that $\mathsf{value} = V$.

To apply Bayes rule, we can read off from the expression that $\mathrm{P}\,[\mathsf{valid} \mid \mathsf{value} = \mathbf{true}] = 0.8$ (or confirm this by calculating the conditional probability explicitly). The other quantities needed for the rule are easily calculated:

- $\mathrm{P}\,[\mathsf{value} = \mathbf{true}] = \mathrm{P}\left[\{\omega_{tt}, \omega_{tf}\}\right] = 0.01$
- $\mathrm{P}\,[\mathsf{valid}] = \mathrm{P}\left[\{\omega_{tt}, \omega_{ft}\}\right] = 0.10304$

Hence, by Bayes rule, the overall semantics is:

$$\mathrm{P}\,[\mathsf{value} = \mathbf{true} \mid \mathsf{valid}] = (0.8 \times 0.01)/0.10304 = 0.07764$$

3 Semantics as Measure Transformers

We cannot generalize the operational semantics of the previous section to continuous distributions, such as **random** (**Gaussian**$(1,1)$), since the probability of any particular sample is zero. A further difficulty is the need to observe events with probability zero, a common situation in machine learning. For example, consider the naive Bayesian classifier, a common, simple probabilistic model. In the training phase, it is given objects together with their classes and the values of their pertinent features. Below, we show the training for a single feature: the weight of the object. The zero probability events are weight measurements, assumed to be normally distributed around the class mean. The outcome of the training is the posterior weight distributions for the different classes.

Naive Bayesian Classifier, Single Feature Training:

```
let wPrior() = random (Gaussian(0.5,1.0))
let Glass,Watch,Plate = wPrior(),wPrior(),wPrior()
let weight objClass objWeight = observe (objWeight−(random (Gaussian(objClass
    ,1.0)))
weight Glass .18; weight Glass .21
weight Watch .11; weight Watch .073
weight Plate .23; weight Plate .45
Watch,Glass,Plate
```

Above, the call to `weight Glass .18` modifies the distribution of the variable `Glass`. The example uses **observe** (`x−y`) to denote that the difference between the weights `x` and `y` is 0. The reason for not instead writing `x=y` is that conditioning on events of zero probability without specifying the random variable they are drawn from is not in general well-defined, cf. Borel's paradox [15]. To avoid this issue, we instead observe the random variable `x−y` of type **real**, at the value 0. (Our compiler does permit the expression **observe** (`x=y`), as sugar for **observe** (`x−y`)).

To give a formal semantics to such observations, as well as to mixtures of continuous and discrete distributions, we turn to measure theory, following standard sources [4,41]. Two basic concepts are measurable spaces and measures. A measurable space is a set of values equipped with a collection of *measurable* subsets; these measurable sets generalize the events of discrete probability. A finite *measure* is a function that assigns a numeric size to each measurable set; measures generalize probability distributions.

We work in the usual mathematical metalanguage of sets and total functions. To machine-check our theory, one might build on a recent formalization of measure theory and Lebesgue integration in higher-order logic [29].

3.1 Types as Measurable Spaces

In the remainder of the paper, we let Ω range over sets of possible outcomes; in our semantics Ω will range over $\mathbb{B} = \{\textbf{true}, \textbf{false}\}$, $\mathbb{Z}$, $\mathbb{R}$, and finite Cartesian products of these sets. A σ-*algebra* over Ω is a set $\mathcal{M} \subseteq \mathcal{P}(\Omega)$ which (1) contains $\varnothing$ and Ω, and (2) is closed under complement and countable union and intersection. A *measurable space*

is a pair $(\Omega, \mathcal{M})$ where $\mathcal{M}$ is a σ-algebra over Ω; the elements of $\mathcal{M}$ are called *measurable sets*. We use the notation $\sigma_\Omega(S)$, when $S \subseteq \mathcal{P}(\Omega)$, for the smallest σ-algebra over Ω that is a superset of S; we may omit Ω when it is clear from context. More formally, $\sigma_\Omega(S)$ is the least set closed under the rules: (1) $S \in \sigma_\Omega(S)$ if $S \subseteq \mathcal{P}(\Omega)$; (2) $(\Omega \setminus A) \in \sigma_\Omega(S)$ if $A \in \sigma_\Omega(S)$; (3) $(\bigcup_{i\in I} A_i) \in \sigma_\Omega(S)$ if $A_i \in \sigma_\Omega(S)$ for all $i \in I$, for countable I. Given two measurable spaces $(\Omega_1, \mathcal{M}_1)$ and $(\Omega_2, \mathcal{M}_2)$, we can compute their product as $(\Omega_1, \mathcal{M}_1) \times (\Omega_2, \mathcal{M}_2) \triangleq (\Omega_1 \times \Omega_2, \sigma_{\Omega_1\times\Omega_2}\{A \times B \mid A \in \mathcal{M}_1, B \in \mathcal{M}_2\})$ If $(\Omega, \mathcal{M})$ and $(\Omega', \mathcal{M}')$ are measurable spaces, then the function $f : \Omega \to \Omega'$ is *measurable* if and only if for all $A \in \mathcal{M}'$, $f^{-1}(A) \in \mathcal{M}$, where the *inverse image* $f^{-1} : \mathcal{P}(\Omega') \to \mathcal{P}(\Omega)$ is given by $f^{-1}(A) \triangleq \{\omega \in \Omega \mid f(\omega) \in A\}$. We write $f^{-1}(x)$ for $f^{-1}(\{x\})$ when $x \in \Omega'$.

We give each first-order type t an interpretation as a measurable space $\mathcal{T}[\![t]\!] \triangleq (\mathbf{V}_t, \mathcal{M}_t)$ below. We write `()` for $\varnothing$, the unit value.

Semantics of Types as Measurable Spaces:

$$\mathcal{T}[\![\textbf{unit}]\!] = (\{\texttt{()}\}, \{\{\texttt{()}\}, \varnothing\}) \qquad \mathcal{T}[\![\textbf{bool}]\!] = (\mathbb{B}, \mathcal{P}(\mathbb{B}))$$
$$\mathcal{T}[\![\textbf{int}]\!] = (\mathbb{Z}, \mathcal{P}(\mathbb{Z})) \qquad \mathcal{T}[\![\textbf{real}]\!] = (\mathbb{R}, \sigma_{\mathbb{R}}(\{[a,b] \mid a,b \in \mathbb{R}\}))$$
$$\mathcal{T}[\![t * u]\!] = (\mathbf{V}_t \times \mathbf{V}_u, \sigma_{\mathbf{V}_t\times\mathbf{V}_u}(\{m \times n \mid m \in \mathcal{M}_t,\ n \in \mathcal{M}_u\}))$$

The set $\sigma_{\mathbb{R}}(\{[a,b] \mid a,b \in \mathbb{R}\})$ in the definition of $\mathcal{T}[\![\textbf{real}]\!]$ is the Borel σ-algebra on the real line, which is the smallest σ-algebra containing all closed (and open) intervals. Below, we write $f : t \to u$ to denote that $f : \mathbf{V}_t \to \mathbf{V}_u$ is measurable, that is, that $f^{-1}(B) \in \mathcal{M}_t$ for all $B \in \mathcal{M}_u$.

3.2 Finite Measures

A *finite measure* μ on a measurable space $(\Omega, \mathcal{M})$ is a function $\mathcal{M} \to \mathbb{R}^+$ that is countably additive, that is, if the sets $A_0, A_1, \ldots \in \mathcal{M}$ are pairwise disjoint, then $\mu(\cup_i A_i) = \sum_i \mu(A_i)$. We write $|\mu| \triangleq \mu(\Omega)$. A consequence of this definition is that $\mu(\varnothing) = 0$. Let $\mathtt{M}\,t$ be the set of finite measures on the measurable space $\mathcal{T}[\![t]\!]$. Additionally, a finite measure μ on $(\Omega, \mathcal{M})$ is a *probability measure* when $|\mu| = 1$. We do not restrict $\mathtt{M}\,t$ to just probability measures, although one can obtain a probability measure from a non-zero finite measure by normalizing with $1/|\mu|$. We make use of the following constructions on measures.

- Given a function $f : t \to u$ and a measure $\mu \in \mathtt{M}\,t$, there is a measure $\mu f^{-1} \in \mathtt{M}\,u$ given by $(\mu f^{-1})(B) \triangleq \mu(f^{-1}(B))$.
- Given a finite measure μ and a measurable set B, we let $\mu|_B(A) \triangleq \mu(A \cap B)$ be the restriction of μ to B.
- We can add two measures on the same set as $(\mu_1 + \mu_2)(A) \triangleq \mu_1(A) + \mu_2(A)$.
- We can multiply a measure by a positive constant as $(r \cdot \mu)(A) \triangleq r \cdot \mu(A)$.
- The (independent) product $(\mu_1 \times \mu_2)$ of two measures is also definable, and satisfies $(\mu_1 \times \mu_2)(A \times B) = \mu_1(A) \cdot \mu_2(B)$. (Existence and uniqueness follows from the Hahn-Kolmogorov theorem.)
- Given a measure μ on the measurable space $\mathcal{T}[\![t]\!]$, a measurable set $A \in \mathcal{M}_t$ and a function $f : t \to \textbf{real}$, we write $\int_A f\,d\mu$ or equivalently $\int_A f(x)\,d\mu(x)$ for standard (Lebesgue) integration. This integration is always well-defined if μ is finite and f is non-negative and bounded from above.

– Given a measure μ on a measurable space $\mathcal{T}[\![t]\!]$ let a function $\dot{\mu} : t \to \textbf{real}$ be a *density* for μ iff $\mu(A) = \int_A \dot{\mu}\, d\lambda$ for all $A \in \mathcal{M}$, where λ is the standard Lebesgue measure on $\mathcal{T}[\![t]\!]$. (We also use λ-notation for functions, but we trust any ambiguity is easily resolved.)

Standard Distributions Given a closed well-typed Fun expression **random** $(D(V))$ of base type b, we define a corresponding finite measure $\mu_{D(V)}$ on measurable space $\mathcal{T}[\![b]\!]$. In the discrete case, we first define probability masses $D(V)\ c$ of single elements, and hence of singleton sets, and then define the measure $\mu_{D(V)}$ as a countable sum.

Masses $D(V)\ c$ and Measures $\mu_{D(V)}$ for Discrete Probability Distributions:

Bernoulli(p) **true** $\triangleq p$	if $0 \le p \le 1$, 0 otherwise
Bernoulli(p) **false** $\triangleq 1-p$	if $0 \le p \le 1$, 0 otherwise
Binomial$(n,p)\ i \triangleq \binom{i}{n} p^i/n!$	if $0 \le p \le 1$, 0 otherwise
DiscreteUniform$(m)\ i \triangleq 1/m$	if $0 \le i < m$, 0 otherwise
Poisson$(l)\ n \triangleq e^{-l} l^n/n!$	if $l, n \ge 0$, 0 otherwise
$\mu_{D(V)}(A) \triangleq \sum_i D(V)\ c_i$	if $A = \bigcup_i \{c_i\}$ for pairwise distinct c_i

In the continuous case, we first define probability densities $D(V)\ r$ at individual elements r. and then define the measure $\mu_{D(V)}$ as an integral. Below, we write $\mathbf{G}$ for the standard Gamma function, which on naturals n satisfies $\mathbf{G}(n) = (n-1)!$.

Densities $D(V)\ r$ and Measures $\mu_{D(V)}$ for Continuous Probability Distributions:

Gaussian$(m,v)\ r \triangleq e^{-(r-m)^2/2v}/\sqrt{2\pi v}$	if $v > 0$, 0 otherwise
Gamma$(s,p)\ r \triangleq r^{s-1} e^{-pr} p^s/\mathbf{G}(s)$	if $r, s, p > 0$, 0 otherwise
Beta$(a,b)\ r \triangleq r^{a-1}(1-r)^{b-1}\mathbf{G}(a+b)/(\mathbf{G}(a)\mathbf{G}(b))$	if $a, b \ge 0$ and $0 \le r \le 1$, 0 otherwise
$\mu_{D(V)}(A) \triangleq \int_A D(V)\, d\lambda$	where λ is the Lebesgue measure on $\mathbb{R}$

The Dirac δ measure is defined on the measurable space $\mathcal{T}[\![b]\!]$ for each base type b, and is given by $\delta_c(A) \triangleq 1$ if $c \in A$, 0 otherwise. We write δ for $\delta_{0.0}$.

The notion of density can be generalized as follows, yielding an unnormalized counterpart to conditional probability. Given a measure μ on $\mathcal{T}[\![t]\!]$ and a measurable function $p : t \to b$, we consider the family of events $p(x) = c$ where c ranges over $\mathbf{V}_b$. We define $\dot{\mu}[A||p = c] \in \mathbb{R}$ (the μ-density at $p = c$ of A) following [9], by:

Conditional Density: $\dot{\mu}[A||p = c]$

$\dot{\mu}[A||p = c] \triangleq \lim_{i\to\infty} \mu(A \cap p^{-1}(B_i))/\int_{B_i} 1\, d\lambda$	if the limit exists
and is the same for all sequences $\{B_i\}$ of closed sets converging regularly to c.	

Where defined, letting $A \in \mathcal{M}_a, B \in \mathcal{M}_b$, conditional density satisfies the equation

$$\int_B \dot{\mu}[A||p = x]\, d(\mu p^{-1})(x) = \mu(A \cap p^{-1}(B)). \tag{1}$$

In particular, we have $\dot{\mu}[A||p=c]=0$ if b is discrete and $\mu(p^{-1}(c))=0$. To show that our definition of conditional density generalizes the notion of density given above, we have that if μ has a continuous density $\dot{\mu}$ on some neighbourhood of $p^{-1}(c)$ then

$$\dot{\mu}[A||p=c]=\int_A \delta_c(p(x))\dot{\mu}(x)\,d\lambda(x).$$

In general, the Radon-Nikodym theorem implies the existence (cf. [4, Ex 33.5]) of a family of finite measures $\dot{\mu}[\cdot||p=c]$ on $\mathcal{T}[\![t]\!]$ satisfying equation (1) above. If $\mu p^{-1}(c)=0$ this is not the case; two versions of $\dot{\mu}[\cdot||p=\cdot]$ may differ on a set B with $\mu p^{-1}(B)=0$. For convenience, we have given an explicit construction that works for many useful cases.

3.3 Measure Transformers

We will now recast some standard theorems of measure theory as a library of combinators, that we will later use to give semantics to probabilistic languages. A *measure transformer* is a function from finite measures to finite measures. We let $t \rightsquigarrow u$ be the set of functions $\mathtt{M}\,t \to \mathtt{M}\,u$. We use the combinators on measure transformers listed below in the formal semantics of our languages. The definitions of these combinators occupy the remainder of this section. We recall that μ denotes a measure and A a measurable set, of appropriate types.

Measure Transformer Combinators:

$\mathtt{pure} \in (t \to u) \to (t \rightsquigarrow u)$
$\ggg \in (t_1 \rightsquigarrow t_2) \to (t_2 \rightsquigarrow t_3) \to (t_1 \rightsquigarrow t_3)$
$\mathtt{choose} \in (t \to (t \rightsquigarrow u)) \to (t \rightsquigarrow u)$
$\mathtt{extend} \in (\mathbf{V}_t \to \mathtt{M}\,u) \to (t \rightsquigarrow (t * u))$
$\mathtt{observe} \in (t \to b) \to (t \rightsquigarrow t)$

Lifting a Function to a Measure Transformer To lift a pure measurable function to a measure transformer, we use the combinator $\mathtt{pure} \in (t \to u) \to (t \rightsquigarrow u)$. Given $f : t \to u$, we let $\mathtt{pure}\ f\ \mu\ A \triangleq \mu f^{-1}(A)$, where μ is a measure on $\mathcal{T}[\![t]\!]$ and A is a measurable set from $\mathcal{T}[\![u]\!]$ (cf. [4, Eqn 13.7]).

Sequential Composition of Measure Transformers To sequentially compose two measure transformers we use standard function composition, defining $\ggg \in (t_1 \rightsquigarrow t_2) \to (t_2 \rightsquigarrow t_3) \to (t_1 \rightsquigarrow t_3)$ as $T \ggg U \triangleq U \circ T$.

Conditional Choice between Measure Transformers The combinator $\mathtt{choose} \in (t \to (t \rightsquigarrow u)) \to (t \rightsquigarrow u)$ makes a conditional choice between measure transformers, if its first argument is measurable and has finite range. Intuitively, $\mathtt{choose}\ K\ \mu$ first splits $\mathbf{V}_t$ into the equivalence classes modulo K. For each equivalence class, we then run the corresponding measure transformer on μ restricted to the class. Finally, the resulting

finite measures are added together, yielding a finite measure. We let $\texttt{choose}\ K\ \mu\ A \triangleq \sum_{T \in \text{range}(K)} T(\mu|_{K^{-1}(T)})(A)$. If the range of K is permitted to be infinite, $\texttt{choose}\ K\ \mu$ might no longer be a finite measure, since the sum may diverge. In particular, if K is a binary choice mapping all elements of B to T_B and all elements of $C = \overline{B}$ to T_C, we have $\texttt{choose}\ K\ \mu\ A = T_B(\mu|_B)(A) + T_C(\mu|_C)(A)$. (In fact, our only uses of `choose` in this paper are in the semantics of conditional expressions in Fun and conditional statements in Imp, and in each case the argument K to `choose` is a binary choice.)

Extending Domain of a Measure The combinator $\texttt{extend} \in (t \to \mathtt{M}\,u) \to (t \rightsquigarrow (t * u))$ extends the domain of a measure using a function yielding measures. It is reminiscent of creating a dependent pair, since the distribution of the second component depends on the value of the first. For `extend` m to be defined, we require that for every $A \in \mathcal{M}_u$, the function $f_A \triangleq \lambda x.m(x)(A)$ is measurable, non-negative and bounded from above. This will always be the case in our semantics for Fun, since we only use the standard distributions for m above. For all of these, all f_A are piece-wise continuous, and thus measurable. We let $\texttt{extend}\ m\ \mu\ AB \triangleq \int_{\mathbf{V}_t} m(x)(\{y \mid (x,y) \in AB\})d\mu(x)$, where we integrate over the first component (call it x) with respect to the measure μ, and the integrand is the measure $m(x)$ of the set $\{y \mid (x,y) \in A\}$ for each x (cf. [4, Ex. 18.20]).

Observation as a Measure Transformer The combinator $\texttt{observe} \in (t \to b) \to (t \rightsquigarrow t)$ conditions a measure over $\mathcal{T}[\![t]\!]$ on the event that an indicator function of type $t \to b$ is zero. Here observation is *unnormalized* conditioning of a measure on an event. We define:

$$\texttt{observe}\ p\ \mu\ A \triangleq \begin{cases} \dot{\mu}[A||p = 0.0] & \text{if } b = \textbf{real} \\ \mu(A \cap p^{-1}(0_b)) & \text{otherwise} \end{cases}$$

As an example, if $p : t \to \textbf{bool}$ is a predicate on values of type t, we have

$$\texttt{observe}\ p\ \mu\ A = \mu(A \cap \{x \mid p(x) = \textbf{true}\}).$$

In the continuous case, if $\mathbf{V}_t = \mathbb{R} \times \mathbb{R}^k$, $p = \lambda(y,x).(y - c)$ and μ has density $\dot{\mu}$ then

$$\texttt{observe}\ p\ \mu\ A = \int_A \mu(y,x)\, d(\delta_c \times \lambda)(y,x) = \int_{\{x \mid (c,x) \in A\}} \dot{\mu}(c,x)\, d\lambda(x).$$

Notice that $\texttt{observe}\ p\ \mu\ A$ can be greater than $\mu(A)$, for which reason we cannot restrict ourselves to transformation of (sub-)probability measures.

3.4 Measure Transformer Semantics of Fun

In order to give a compositional denotational semantics of Fun programs, we give a semantics to open programs, later to be placed in some closing context. Since observations change the distributions of program variables, we may draw a parallel to while programs. There, a program can be given a denotation as a function from variable valuations to a return value and a variable valuation. Similarly, we give semantics to an open Fun term by mapping a measure over assignments to the term's free variables to a joint

measure of the term's return value and assignments to its free variables. This choice is a generalization of the (discrete) semantics of pWHILE [3]. This contrasts with Ramsey and Pfeffer [39], where the semantics of an open program takes a variable valuation and returns a (monadic computation yielding a) distribution of return values.

First, we define a data structure for an evaluation environment assigning values to variable names, and corresponding operations. Given an environment $\Gamma = x_1{:}t_1, \ldots, x_n{:}t_n$, we let $\mathtt{S}\langle\Gamma\rangle$ be the set of states, or finite maps $s = \{x_1 \mapsto c_1, \ldots, x_n \mapsto c_n\}$ such that for all $i = 1, \ldots, n$, $\mathrm{ty}(c_i) = t_i$. We let $\mathcal{T}[\![\mathtt{S}\langle\Gamma\rangle]\!] \triangleq \mathcal{T}[\![t_1 * \cdots * t_n]\!]$ be the measurable space of states in $\mathtt{S}\langle\Gamma\rangle$. We define $\mathrm{dom}(s) \triangleq \{x_1, \ldots, x_n\}$. We define the following operators.

Auxiliary Operations on States and Pairs:

$\mathtt{add}\ x\ (s,c) \triangleq s \cup \{x \mapsto c\}$	if $\mathrm{ty}(c) = t$ and $x \notin \mathrm{dom}(s)$, s otherwise.
$\mathtt{lookup}\ x\ \ s \triangleq s(x)$	if $x \in \mathrm{dom}(s)$, $()$ otherwise.
$\mathtt{drop}\ X\ s \triangleq \{(x \mapsto c) \in s \mid x \notin X\}$	$\mathtt{fst}((x,y)) \triangleq x \qquad \mathtt{snd}((x,y)) \triangleq y$

We write $s|_X$ for $\mathtt{drop}\ (\mathrm{dom}(s) \setminus X)\ s$.We apply these combinators to give a semantics to Fun programs as measure transformers. We assume that all bound variables in a program are different from the free variables and each other. Below, $\mathcal{V}[\![V]\!]\ s$ gives the valuation of V in state s, and $\mathcal{A}[\![M]\!]$ gives the measure transformer denoted by M.

Measure Transformer Semantics of Fun:

$$\mathcal{V}[\![x]\!]\ s \triangleq \mathtt{lookup}\ x\ s$$
$$\mathcal{V}[\![c]\!]\ s \triangleq c$$
$$\mathcal{V}[\![(V_1, V_2)]\!]\ s \triangleq (\mathcal{V}[\![V_1]\!]\ s, \mathcal{V}[\![V_2]\!]\ s)$$

$$\mathcal{A}[\![V]\!] \triangleq \mathtt{pure}\ \lambda s.(s, \mathcal{V}[\![V]\!]\ s)$$
$$\mathcal{A}[\![V_1 \otimes V_2]\!] \triangleq \mathtt{pure}\ \lambda s.(s, ((\mathcal{V}[\![V_1]\!]\ s) \otimes (\mathcal{V}[\![V_2]\!]\ s)))$$
$$\mathcal{A}[\![V.1]\!] \triangleq \mathtt{pure}\ \lambda s.(s, \mathtt{fst}(\mathcal{V}[\![V]\!]\ s))$$
$$\mathcal{A}[\![V.2]\!] \triangleq \mathtt{pure}\ \lambda s.(s, \mathtt{snd}(\mathcal{V}[\![V]\!]\ s))$$

$$\mathcal{A}[\![\textbf{if}\ V\ \textbf{then}\ M\ \textbf{else}\ N]\!] \triangleq \mathtt{choose}\ \lambda s.\mathtt{if}\ \mathcal{V}[\![V]\!]\ s\ \mathtt{then}\ \mathcal{A}[\![M]\!]\ \mathtt{else}\ \mathcal{A}[\![N]\!]$$
$$\mathcal{A}[\![\textbf{random}\ (D(V))]\!] \triangleq \mathtt{extend}\ \lambda s.\mu_{D(\mathcal{V}[\![V]\!]\ s)}$$
$$\mathcal{A}[\![\textbf{observe}\ V]\!] \triangleq (\mathtt{observe}\ \lambda s.\mathcal{V}[\![V]\!]\ s) \ggg \mathtt{pure}\ \lambda s.(s, ())$$
$$\mathcal{A}[\![\textbf{let}\ x = M\ \textbf{in}\ N]\!] \triangleq$$
$$\quad \mathcal{A}[\![M]\!] \ggg \mathtt{pure}\ (\mathtt{add}\ x) \ggg \mathcal{A}[\![N]\!] \ggg \mathtt{pure}\ \lambda(s,y).((\mathtt{drop}\ \{x\}\ s), y)$$

A value expression V returns the valuation of V in the current state, which is left unchanged. Similarly, binary operations and projections have a deterministic meaning given the current state. An **if** V expression runs the measure transformer given by the **then** branch on the states where V evaluates true, and the transformer given by the **else** branch on all other states, using the combinator `choose`. A primitive distribution **random** $(D(V))$ extends the state measure with a value drawn from the distribution D, with parameters V depending on the current state. An observation **observe** V modifies the current measure by restricting it to states where V is zero. It is implemented with the `observe` combinator, and it always returns the unit value. The expression **let** $x = M$ **in** N

intuitively first runs M and binds its return value to x using `add`. After running N, the binding is discarded using `drop`.

Lemma 8. *If* $s : \mathtt{S}\langle\Gamma\rangle$ *and* $\Gamma \vdash V : t$ *then* $\mathcal{V}[\![V]\!]\, s \in \mathbf{V}_t$.

Lemma 9. *If* $\Gamma \vdash M : t$ *then* $\mathcal{A}[\![M]\!] \in \mathtt{S}\langle\Gamma\rangle \rightsquigarrow (\mathtt{S}\langle\Gamma\rangle * t)$.

The measure transformer semantics of Fun is hard to use directly, except in the case of Bernoulli Fun where they can be directly implemented: a naive implementation of $\mathtt{M}\langle\mathtt{S}\langle\Gamma\rangle\rangle$ is as a map assigning a probability to each possible variable valuation. If there are N variables, each sampled from a Bernoulli distribution, in the worst case there are 2^N paths to be explored in the computation, each of which corresponds to a variable valuation. Our direct implementation of the measure transformer semantics, described in Appendix B, explicitly constructs the valuation. It works fine for small examples but would blow up on large datasets. In this simple case, the measure transformer semantics of closed programs also coincides with the sampling semantics.

Theorem 1. *Suppose* $\varepsilon \vdash M : t$ *for some* M *in Bernoulli Fun. If* $\mu = \mathcal{A}[\![M]\!]\ \delta_{()}$ *and* $\varepsilon \vdash V : t$ *then* $\mathrm{P}_M\,[\mathsf{value} = V \mid \mathsf{valid}] = \mu(\{V\})/|\mu|$.

Proof. In order to generalize the semantics of Bernoulli Fun to measure transformers, we consider open programs M starting in an initial state measure $\mu \in \mathtt{M}\langle\mathtt{S}\langle\Gamma\rangle\rangle$, written $\mathbf{init}(M,\mu)$. We let $\mathbf{init}(M,\mu) \to^{p_s} M\{{}^{V_1}/_{x_1} \cdots {}^{V_n}/_{x_n}\}$ when $s = \{x_i \mapsto V_i \mid i = 1..n\} \in \mathtt{S}\langle\Gamma\rangle$ and $p_s = \mu(\{s' \mid s'|_{\mathrm{fv}(M)} = s|_{\mathrm{fv}(M)}\})$.

If $\Gamma \vdash M : t$, $\varepsilon \vdash V : t$ and $\mu \in \mathtt{M}\langle\mathtt{S}\langle\Gamma\rangle\rangle$, we let $\nu = \mathcal{A}[\![M]\!]\ \mu$. Then $\nu(\{V\}) = \mathrm{P}\,[\mathsf{valid} \cap \mathsf{value} = V]$ and $\nu(\mathbf{V}_t) = \mathrm{P}\,[\mathsf{valid}]$, where P is relative to the program $\mathbf{init}(M,\mu)$. The proof is by induction on the derivation of $\Gamma \vdash M : t$.

Then, $\mathrm{P}\,[\mathsf{value} = V \mid \mathsf{valid}] = \mathrm{P}\,[\mathsf{valid} \cap \mathsf{value} = V]\,/\mathrm{P}\,[\mathsf{valid}] = \nu(\{V\})/\nu(\mathbf{V}_t)$. □

3.5 Discussion of the Semantics

Discrete Observations amount to filtering. A consequence of Theorem 1 is that our measure transformer semantics is a generalization of the sampling semantics for discrete probabilities. For this theorem to hold, it is critical that `observe` denotes unnormalized conditioning (filtering). Otherwise programs that perform observations inside the branches of conditional expressions would have undesired semantics. As the following example shows, the two program fragments **observe** (x=y) and **if** x **then observe** (y=**true**) **else observe** (y=**false**) would have different measure transformer semantics although they have the same sampling semantics.

Simple Conditional Expression: M_{if}

```
let x = random (Bernoulli(0.5))
let y = random (Bernoulli(0.1))
if x then observe (y=true) else observe (y=false)
y
```

In the sampling semantics, the two valid runs are when x and y are both **true** (with probability 0.05), and both **false** (with probability 0.45), so we have $\mathrm{P}\,[\textbf{true} \mid \mathsf{valid}] = 0.1$ and $\mathrm{P}\,[\textbf{false} \mid \mathsf{valid}] = 0.9$.

If, instead of the unnormalized definition $\texttt{observe}\ p\ \mu\ A = \mu(A \cap \{x \mid p(x)\})$, we had either of the normalizing definitions

$$\texttt{observe}\ p\ \mu\ A = \frac{\mu(A \cap \{x \mid p(x)\})}{\mu(\{x \mid p(x)\})} \qquad \text{or} \qquad |\mu|\frac{\mu(A \cap \{x \mid p(x)\})}{\mu(\{x \mid p(x)\})}$$

then $\mathcal{A}[\![M_{\mathrm{if}}]\!]\ \delta_{()}\ \{\textbf{true}\} = \mathcal{A}[\![M_{\mathrm{if}}]\!]\ \delta_{()}\ \{\textbf{false}\}$, which would invalidate the theorem.

Let $M' = M_{\mathrm{if}}$ with **observe** $(x = y)$ substituted for the conditional expression. With the actual or either of the flawed definitions of `observe` we have $\mathcal{A}[\![M']\!]\ \delta_{()}\ \{\textbf{true}\} = (\mathcal{A}[\![M']\!]\ \delta_{()}\ \{\textbf{false}\})/9$.

Discrete versus continuous observations. As an example to highlight the difference between continuous and discrete observations, we first consider the following program, which observes that a normally distributed random variable is zero. The resulting distribution of the variable is a point mass at 0.0, as expected.

Continuous Observation:

```
let x = random (Gaussian(0.0, 1.0)) in let _ = observe x in x
```

The second program instead observes that a Boolean variable is true. This has zero probability of occurring, and since the Boolean type is discrete, the resulting measure is the zero measure.

Discrete Observation:

```
let x = random (Gaussian(0.0, 1.0)) in let b = (x==0.0) in let _ = observe b in x
```

These examples show the need for observations at **real** type, as well as at type **bool**. (This also clearly distinguishes **observe** from assume in assertional programming.)

Medical trial. As another example, let us consider a simple Bayesian evaluation of a medical trial [30]. We assume a trial group of nTrial persons, of which cTrial were healthy at the end of the trial, and a control group of nControl persons, of which cControl were healthy at the end of the trial. Below, **Beta**(1.0,1.0) is the uniform distribution on the interval $[0.0, 1.0]$. We return the posterior distributions of the likelihood that a member of the trial group (pTrial) and a member of the control group (pControl) is healthy at the end of the trial.

Medical Trial:

```
let medicalTrial nTrial nControl cTrial cControl =
  let pTrial = random(Beta(1.0,1.0))
  observe (cTrial == random (Binomial(nTrial,pTrial)));
  let pControl = random(Beta(1.0,1.0))
  observe (cControl == random (Binomial(nControl,pControl)));
  pTrial, pControl
```

We can then compare this model to one where the treatment is ineffective, that is, where the members of the trial group and the control group have the same probability of becoming healthy. Also here we give a uniform prior to the probability that the treatment is effective; the posterior distribution of this variable will depend on the Bayesian evidence for the different models, that is, the ratio between the probabilities of the observed outcome in the two models.

Model Selection:

```
let modelSelection nTrial nControl cTrial cControl =
  let pEffective = random(Beta(1.0,1.0))
  if random(Bernoulli(pEffective)) then
    medicalTrial nTrial nControl cTrial cControl
    ()
  else
    let pAll = random(Beta(1.0,1.0))
    observe (cTrial == random (Binomial(nTrial,pAll)))
    observe (cControl == random (Binomial(nControl,pAll)))
  pEffective
```

4 Semantics as Factor Graphs

A naive implementation of the measure transformer semantics of the previous section would work directly with measures of states, whose size even in the discrete case could be exponential in the number of variables in scope. For large models, this becomes intractable. In this section, we instead give a semantics to Fun programs as *factor graphs* [22], whose size will be linear in the size of the program. We define this semantics in two steps. We first compile the Fun program into a program in the simple imperative language Imp, and then the Imp program itself has a straightforward semantics as a factor graph. Our semantics formalizes the way in which our implementation maps F# programs to Csoft programs, which are evaluated by Infer.NET by constructing suitable factor graphs. The implementation advantage of translating F# to Csoft, over simply generating factor graphs directly [26], is that the translation preserves the structure of the input model (including array processing in our full language), which can be exploited by the various inference algorithms supported by Infer.NET.

4.1 Imp: An Imperative Core Calculus

Imp is an imperative language, based on the static single assignment (SSA) intermediate form. It is a sublanguage of Csoft, the input language of Infer.NET [30], and is intended to have a simple semantics as a factor graph. A composite statement C is a sequence of statements, each of which either stores the result of a primitive operation in a location, observes the contents of a location to be zero, or branches on the value of a location. Imp shares the base types b with Fun, but has no tuples.

Syntax of Imp:

$l, l', \ldots$	location (variable) in global store
$E, F ::= c \mid l \mid (l \otimes l)$	expression
$I ::=$	statement
$\quad l \leftarrow E$	assignment
$\quad l \xleftarrow{s} D(l_1, \ldots, l_n)$	random assignment
$\quad \mathbf{observe}_b\ l$	observation
$\quad \mathbf{if}\ l\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2$	conditional
$\quad \mathbf{local}\ l : b\ \mathbf{in}\ C$	local declaration (scope of l is C)
$C ::= \mathbf{nil} \mid I \mid (C; C)$	composite statement

When making an observation $\mathbf{observe}_b$, we make explicit the type b of the observed location. In a local declaration, $\mathbf{local}\ l : b\ \mathbf{in}\ C$, the location l is bound, with scope C.

Next, we derive an extended form of **local**, which introduces a sequence of local variables. We also derive an extended conditional, which binds local variables separately in its then and else branch. (This extended conditional is the only way to introduce local variables in the conference version of this article; in this full version, separating the two features leads to a simpler semantics.)

Some Derived Forms:

$\mathbf{local}\ \Sigma\ \mathbf{in}\ C \triangleq \mathbf{local}\ l_1 : b_1\ \mathbf{in}\ \ldots \mathbf{local}\ l_n : b_n\ \mathbf{in}\ C \quad$ where $\Sigma = \varepsilon, l_1 : b_1, \ldots, l_n : b_n$

$\mathbf{if}\ l\ \mathbf{then}_{\Sigma_1}\ C_1\ \mathbf{else}_{\Sigma_2}\ C_2 \triangleq \mathbf{if}\ l\ \mathbf{then}\ (\mathbf{local}\ \Sigma_1\ \mathbf{in}\ C_1)\ \mathbf{else}\ (\mathbf{local}\ \Sigma_2\ \mathbf{in}\ C_2)$

The typing rules for Imp are standard. We consider Imp typing environments Σ to be a special case of Fun environments Γ, where variables (locations) always map to base types. If $\Sigma = \varepsilon, l_1 : b_1, \ldots, l_n : b_n$, we say Σ is *well-formed* and write $\Sigma \vdash \diamond$ to mean that the locations l_i are pairwise distinct. The judgment $\Sigma \vdash E : b$ means that the expression E has type b in the environment Σ. The judgment $\Sigma \vdash C : \Sigma'$ means that the composite statement C is well-typed in the initial environment Σ, yielding additional bindings Σ'.

Judgments of the Imp Type System:

$\Sigma \vdash \diamond$	environment Σ is well-formed
$\Sigma \vdash E : b$	in Σ, expression E has type b
$\Sigma \vdash C : \Sigma'$	given Σ, statement C assigns to Σ'

Typing Rules for Imp Expressions and Commands:

(IMP CONST)
$$\frac{\Sigma \vdash \diamond}{\Sigma \vdash c : \mathrm{ty}(c)}$$

(IMP LOC)
$$\frac{\Sigma \vdash \diamond \quad (l{:}b) \in \Sigma}{\Sigma \vdash l : b}$$

(IMP OP)
$$\frac{\Sigma \vdash l_1 : b_1 \quad \Sigma \vdash l_2 : b_2 \quad \otimes : b_1, b_2 \to b_3}{\Sigma \vdash l_1 \otimes l_2 : b_3}$$

(IMP ASSIGN)
$$\frac{\Sigma \vdash E : b \qquad l \notin \mathrm{dom}(\Sigma)}{\Sigma \vdash l \leftarrow E : (\varepsilon, l{:}b)}$$

(IMP RANDOM)
$$\frac{D : (x_1 : b_1, \ldots, x_n : b_n) \to b \qquad l \notin \mathrm{dom}(\Sigma) \qquad \Sigma \vdash l_1 : b_1 \quad \cdots \quad \Sigma \vdash l_n : b_n}{\Sigma \vdash l \xleftarrow{s} D(l_1, \ldots, l_n) : (\varepsilon, l{:}b)}$$

$$\frac{}{\Sigma \vdash \diamond}\ (\textsc{Imp Nil})\qquad \frac{\Sigma \vdash l : b}{\Sigma \vdash \mathbf{observe}_b\ l : \varepsilon}\ (\textsc{Imp Observe})\qquad \frac{\Sigma \vdash C_1 : \Sigma' \qquad \Sigma, \Sigma' \vdash C_2 : \Sigma''}{\Sigma \vdash C_1; C_2 : \Sigma', \Sigma''}\ (\textsc{Imp Seq})$$

$$\frac{\Sigma \vdash l : \mathbf{bool} \quad \Sigma \vdash C_1 : \Sigma' \quad \Sigma \vdash C_2 : \Sigma'}{\Sigma \vdash \mathbf{if}\ l\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2 : \Sigma'}\ (\textsc{Imp If})\qquad \frac{\Sigma \vdash C : \Sigma' \quad (l : b) \in \Sigma'}{\Sigma \vdash \mathbf{local}\ l : b\ \mathbf{in}\ C : (\Sigma' \setminus \{l : b\})}\ (\textsc{Imp Local})$$

We define order-preserving merging of typing environments as follows: $\Sigma \in \Sigma_1 + \Sigma_2$ if either $\Sigma = \Sigma_1 = \Sigma_2 = \varepsilon$, or there are $i \in \{1,2\}, \Sigma', \Sigma_i', x, b$ such that $\Sigma_i = (\Sigma_i', x : b)$ and $\Sigma = (\Sigma', x : b)$ and $\Sigma' \in \Sigma_i' + \Sigma_{3-i}$.

Typing Rules for Derived Forms:

$$\frac{\Sigma \vdash C : \Sigma_1' \quad \Sigma_1' \in \Sigma_1 + \Sigma'}{\Sigma \vdash \mathbf{local}\ \Sigma_1\ \mathbf{in}\ C : \Sigma'}\ (\textsc{Imp Locals})$$

$$\frac{\Sigma \vdash l : \mathbf{bool} \quad \Sigma \vdash C_1 : \Sigma_1' \quad \Sigma \vdash C_2 : \Sigma_2' \quad \Sigma_1' \in \Sigma_1 + \Sigma' \quad \Sigma_2' \in \Sigma_2 + \Sigma'}{\Sigma \vdash \mathbf{if}\ l\ \mathbf{then}_{\Sigma_1}\ C_1\ \mathbf{else}_{\Sigma_2}\ C_2 : \Sigma'}\ (\textsc{Imp If Locals})$$

Lemma 10.

(1) *If* $\Sigma, \Sigma' \vdash \diamond$ *then* $\mathrm{dom}(\Sigma) \cap \mathrm{dom}(\Sigma') = \varnothing$.
(2) *If* $\Sigma \vdash E : b$ *then* $\Sigma \vdash \diamond$ *and* $\mathrm{fv}(E) \subseteq \mathrm{dom}(\Sigma)$.
(3) *If* $\Sigma \vdash C : \Sigma'$ *then* $\Sigma, \Sigma' \vdash \diamond$.

4.2 Measure Transformer Semantics of Imp

A compound statement C in Imp has a semantics as a measure transformer $\mathcal{I}[\![C]\!]$ generated from the set of combinators defined in Section 3. An Imp program does not return a value, but is solely a measure transformer on states $\mathtt{S}\langle\Sigma\rangle$ (where Σ is a special case of Γ).

Interpretation of Statements: $\mathcal{I}[\![C]\!], \mathcal{I}[\![I]\!] : \mathtt{S}\langle\Sigma\rangle \rightsquigarrow \mathtt{S}\langle\Sigma'\rangle$

$$\mathcal{I}[\![\mathbf{nil}]\!] \triangleq \mathtt{pure}\ \mathtt{id}$$
$$\mathcal{I}[\![C_1; C_2]\!] \triangleq \mathcal{I}[\![C_1]\!] \ggg \mathcal{I}[\![C_2]\!]$$
$$\mathcal{I}[\![l \leftarrow c]\!] \triangleq \mathtt{pure}\ \lambda s.\mathtt{add}\ l\ (s, c)$$
$$\mathcal{I}[\![l \leftarrow l']\!] \triangleq \mathtt{pure}\ \lambda s.\mathtt{add}\ l\ (s, \mathtt{lookup}\ l'\ s)$$
$$\mathcal{I}[\![l \leftarrow l_1 \otimes l_2]\!] \triangleq \mathtt{pure}\ \lambda s.\mathtt{add}\ l\ (s, (\mathtt{lookup}\ l_1\ s \otimes \mathtt{lookup}\ l_2\ s)))$$
$$\mathcal{I}[\![l \overset{s}{\leftarrow} D(l_1, \ldots, l_n)]\!] \triangleq \mathtt{extend}\ (\lambda s.\mu_{D(\mathtt{lookup}\ l_1\ s, \ldots, \mathtt{lookup}\ l_n\ s)}) \ggg \mathtt{pure}\ (\mathtt{add}\ l)$$
$$\mathcal{I}[\![\mathbf{observe}_b\ l]\!] \triangleq \mathtt{observe}\ \lambda s.\mathtt{lookup}\ l\ s$$
$$\mathcal{I}[\![\mathbf{if}\ l\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2]\!] \triangleq \mathtt{choose}\ \lambda s.\mathtt{if}\ (\mathtt{lookup}\ l\ s)\ \mathtt{then}\ \mathcal{I}[\![C_1]\!]\ \mathtt{else}\ \mathcal{I}[\![C_2]\!]$$
$$\mathcal{I}[\![\mathbf{local}\ l : b\ \mathbf{in}\ C]\!] \triangleq \mathcal{I}[\![C]\!] \ggg \mathtt{pure}\ (\mathtt{drop}\ \{l\})$$

The main difference to the semantics of Fun is that Imp programs do not return values.

Semantics of Derived Forms:

$\mathcal{I}[\![\textbf{local } \Sigma \textbf{ in } C]\!] \triangleq \mathcal{I}[\![C]\!] \ggg \texttt{pure}\ (\texttt{drop}\ (\mathrm{dom}\,\Sigma))$
$\mathcal{I}[\![\textbf{if } l \textbf{ then}_{\Sigma_1} C_1 \textbf{ else}_{\Sigma_2} C_2]\!] \triangleq \texttt{choose}\ \lambda s.\texttt{if}\ (\texttt{lookup}\ l\ s)$
$\texttt{then}\ (\mathcal{I}[\![C_1]\!] \ggg \texttt{pure}\ (\texttt{drop}\ (\mathrm{dom}\,\Sigma_1)))\ \texttt{else}\ (\mathcal{I}[\![C_2]\!] \ggg \texttt{pure}\ (\texttt{drop}\ (\mathrm{dom}\,\Sigma_2)))$

4.3 Translating from Fun to Imp

The translation from Fun to Imp is a mostly routine compilation of functional code to imperative code. The main point of interest is that Imp locations only hold values of base type, while Fun variables may hold tuples. We rely on *patterns* p and *layouts* ρ to track the Imp locations corresponding to Fun environments.

Notations for the Translation from Fun to Imp:

$p ::= l \mid () \mid (p,p)$	pattern: group of Imp locations to represent Fun value
$\rho ::= (x_i \mapsto p_i)^{i\in 1..n}$	layout: finite map from Fun variables to patterns
$\Sigma \vdash p : t$	in environment Σ, pattern p represents Fun value of type t
$\Sigma \vdash \rho : \Gamma$	in environment Σ, layout ρ represents environment Γ
$\rho \vdash M \Rightarrow C, p$	given ρ, expression M translates to C and pattern p

Rules for Patterns $\Sigma \vdash p : t$ and Layouts $\Sigma \vdash \rho : \Gamma$:

(PAT LOC) $\dfrac{\Sigma \vdash \diamond \quad (l:t) \in \Sigma}{\Sigma \vdash l : t}$

(PAT UNIT) $\dfrac{\Sigma \vdash \diamond}{\Sigma \vdash () : \textbf{unit}}$

(PAT PAIR) $\dfrac{\Sigma \vdash p_1 : t_1 \quad \Sigma \vdash p_2 : t_2}{\Sigma \vdash (p_1, p_2) : t_1 * t_2}$

(LAYOUT) $\dfrac{\mathrm{locs}(\rho) = \mathrm{dom}(\Sigma) \quad \Sigma \vdash \diamond \quad \mathrm{dom}(\rho) = \mathrm{dom}(\Gamma) \quad \Sigma \vdash \rho(x) : t \quad \forall (x:t) \in \Gamma}{\Sigma \vdash \rho : \Gamma}$

The rule (PAT LOC) represents values of base type by a single location. The rules (PAT UNIT) and (LAYOUT) represent products by a pattern for their corresponding components. The rule (LAYOUT) asks that each entry in Γ is assigned a pattern of suitable type by layout ρ.

The translation rules below depend on some additional notations. Let $p \sim p'$ be the congruence closure on patterns of the total relation on locations, so that $p \sim p'$ means that p and p' are patterns with the same shape. We write $p \leftarrow p'$ for piecewise assignment of $p \sim p'$. We say $p \in \Sigma$ if every location in p is in Σ. Let $\mathrm{locs}(\rho) = \bigcup\{\mathrm{fv}(\rho(x)) \mid x \in \mathrm{dom}(\rho)\}$, and let $\mathrm{locs}(C)$ be the environment listing the set of locations assigned by a command C.

Rules for Translation: $\rho \vdash M \Rightarrow C, p$

(TRANS VAR) $\dfrac{}{\rho \vdash x \Rightarrow \textbf{nil}, \rho(x)}$

(TRANS CONST) $\dfrac{c \neq () \quad l \notin \mathrm{locs}(\rho)}{\rho \vdash c \Rightarrow (l \leftarrow c), l}$

(TRANS UNIT) $\dfrac{}{\rho \vdash () \Rightarrow \textbf{nil}, ()}$

(TRANS OPERATOR)

$$\frac{\rho \vdash V_1 \Rightarrow C_1, l_1 \qquad \rho \vdash V_2 \Rightarrow C_2, l_2 \qquad l \notin \mathrm{locs}(\rho) \cup \mathrm{locs}(C_1) \cup \mathrm{locs}(C_2) \qquad \mathrm{locs}(C_1) \cap \mathrm{locs}(C_2) = \varnothing}{\rho \vdash V_1 \otimes V_2 \Rightarrow (C_1; C_2; l \leftarrow l_1 \otimes l_2), l}$$

(TRANS PAIR)

$$\frac{\rho \vdash V_1 \Rightarrow C_1, p_1 \quad \rho \vdash V_2 \Rightarrow C_2, p_2 \quad \mathrm{locs}(C_1) \cap \mathrm{locs}(C_2) = \varnothing}{\rho \vdash (V_1, V_2) \Rightarrow (C_1; C_2), (p_1, p_2)}$$

(TRANS PROJ1)

$$\frac{\rho \vdash V \Rightarrow C, (p_1, p_2)}{\rho \vdash V.1 \Rightarrow C, p_1}$$

(TRANS PROJ2)

$$\frac{\rho \vdash V \Rightarrow C, (p_1, p_2)}{\rho \vdash V.2 \Rightarrow C, p_2}$$

(TRANS IF)

$$\frac{\begin{array}{l}\rho \vdash V_1 \Rightarrow C_1, l \quad (\mathrm{locs}(\rho) \cup \mathrm{locs}(C_1) \cup \mathrm{locs}(C_2) \cup \mathrm{locs}(C_3)) \cap \mathrm{fv}(p) = \varnothing \\ \rho \vdash M_2 \Rightarrow C_2, p_2 \quad C_2' = \textbf{local } \mathrm{locs}(C_2) \textbf{ in } (C_2; p \leftarrow p_2) \quad p_2 \sim p \\ \rho \vdash M_3 \Rightarrow C_3, p_3 \quad C_3' = \textbf{local } \mathrm{locs}(C_3) \textbf{ in } (C_3; p \leftarrow p_3) \quad p_3 \sim p\end{array}}{\rho \vdash (\textbf{if } V_1 \textbf{ then } M_2 \textbf{ else } M_3) \Rightarrow (C_1; \textbf{if } l \textbf{ then } C_2' \textbf{ else } C_3'), p}$$

(TRANS OBSERVE)

$$\frac{\rho \vdash V \Rightarrow C, l \quad b \text{ is the type of } V}{\rho \vdash \textbf{observe } V \Rightarrow (C; \textbf{observe}_b\ l), ()}$$

(TRANS RANDOM)

$$\frac{\rho \vdash V \Rightarrow C, p \quad l \notin \mathrm{locs}(\rho) \cup \mathrm{locs}(C)}{\rho \vdash \textbf{random } (D(V)) \Rightarrow (C; l \xleftarrow{s} D(p)), l}$$

(TRANS LET)

$$\frac{\rho \vdash M_1 \Rightarrow C_1, p_1 \quad x \notin \mathrm{dom}(\rho) \quad \rho\{x \mapsto p_1\} \vdash M_2 \Rightarrow C_2, p_2 \quad \mathrm{locs}(C_1) \cap \mathrm{locs}(C_2) = \varnothing}{\rho \vdash \textbf{let } x = M_1 \textbf{ in } M_2 \Rightarrow (C_1; C_2), p_2}$$

In general, a Fun term M translates under a layout ρ to a series of commands C and a pattern p. The commands C mutate the global store so that the locations in p correspond to the value that M returns. The simplest example of this is in (TRANS CONST): the constant expression c translates to an Imp program that writes c into a fresh location l. The pattern that represents this return value is l itself. The (TRANS VAR) and (TRANS UNIT) rules are similar. In both rules, no commands are run. For variables, we look up the pattern in the layout ρ; for unit, we return the unit location. Translation of pairs (TRANS PAIR) builds each of the constituent values and constructs a new pair pattern.

More interesting are the projection operators. Consider (TRANS PROJ1); the second projection is translated similarly by (TRANS PROJ2). To find $V.1$, we run the commands to generate V, which we know must return a pair pattern (p_1, p_2). To extract the first element of this pair, we simply need to return p_1. Not only would it not be easy to isolate and run only the commands to generate the values that go in p_1, it would be incorrect to do so. For example, the Fun expressions constructing the second element of V may observe values, and hence have non-local effects.

The translation for conditionals (TRANS IF) is somewhat subtle. First, the conditional and branches are translated, and the commands to generate the result of conditional are run before the test itself. Next, a pattern p of fresh locations is used to hold the return value ; using a shared output pattern allows us to avoid the ϕ nodes common in SSA compilers. (We arbitrarily require that $p_2 \sim p$, which implies $p_3 \sim p$.) Finally, we

use the Imp derived form where the **then** and **else** branches of a conditional are marked with environments for their local variables. In this case, these environments list all of the locations written in the branch except for the locations in the shared target p.

The rule (TRANS OBSERVE) translates **observe** by running the commands to generate the value for V and then observing the pattern. (This pattern l can only be a location, and not of the form $()$ or (p_1, p_2), as observations are only possible on values of base type.)

The rule (TRANS RANDOM) translates random sampling in much the same way. By $D(p)$, we mean the flattening of p into a list of locations and passing it to the distribution constructor D.

Finally, the rule (TRANS LET) translates **let** statements by running both expressions in sequence. We translate M_2, the body of the let, with an extended layout, so that C_2 knows where to find the values written by C_1, in the pattern p_1.

Proposition 2. *Suppose $\Gamma \vdash M : t$ and $\Sigma \vdash \rho : \Gamma$.*

(1) *There are C and p such that $\rho \vdash M \Rightarrow C, p$.*
(2) *Whenever $\rho \vdash M \Rightarrow C, p$, there is Σ' such that $\Sigma \vdash C : \Sigma'$ and $\Sigma, \Sigma' \vdash p : t$.*

Proof. By induction on the typing of M (Appendix A.1). □

We define operations `lift` and `restrict` to translate between Fun variables ($\mathtt{S}\langle\Gamma\rangle$) and Imp locations ($\mathtt{S}\langle\Sigma\rangle$).

$$\mathtt{lift}\ \rho \triangleq \lambda s.\, \mathrm{flatten}\,\{\rho(x) \mapsto \mathcal{V}[\![x]\!]\ s \mid x \in \mathrm{dom}(\rho)\}$$
$$\mathtt{restrict}\ \rho \triangleq \lambda s.\, \{x \mapsto \mathcal{V}[\![\rho(x)]\!]\ s \mid x \in \mathrm{dom}(\rho)\}$$

We let flatten take a mapping from patterns to values to a mapping from locations to base values. Given these notations, we state that the compilation of Fun to Imp preserves the measure transformer semantics, modulo a pattern p that indicates the locations of the various parts of the return value in the typing environment; an environment mapping ρ, which does the same translation for the initial typing environment; and superfluous variables, removed by `restrict`.

Theorem 2. *If $\Gamma \vdash M : t$ and $\Sigma \vdash \rho : \Gamma$ and $\rho \vdash M \Rightarrow C, p$ then:*
$\mathcal{A}[\![M]\!] = \mathtt{pure}\ (\mathtt{lift}\ \rho) \ggg \mathcal{I}[\![C]\!] \ggg \mathtt{pure}\ (\lambda s.\ (\mathtt{restrict}\ \rho\ s, p\ s))$.

Proof. By induction on the typing of M (Appendix A.2). □

4.4 Factor Graphs

A factor graph [22] represents a joint probability distribution of a set of random variables as a collection of multiplicative factors. Factor graphs are an effective means of stating conditional independence properties between variables, and enable efficient algebraic inference techniques [32,45] as well as sampling techniques [19, Chapter 12]. We use factor graphs with *gates* [31] for modelling if-then-else clauses; gates introduce second-order edges in the graph.

Factor Graphs:

$G ::= \mathsf{new}\ \overline{x : b}\ \mathsf{in}\ \{e_1, \ldots, e_m\}$	graph (variables $\overline{x}$ distinct, bound in $e_1, \ldots, e_m$)
$x, y, z, \ldots$	nodes (random variables)
$e ::=$	edge
$\quad \mathsf{Equal}(x, y)$	$\quad$ equality ($x = y$)
$\quad \mathsf{Constant}_c(x)$	$\quad$ constant ($x = c$)
$\quad \mathsf{Binop}_{\otimes}(x, y, z)$	$\quad$ binary operator ($x = y \otimes z$)
$\quad \mathsf{Sample}_D(x, y_1, \ldots, y_n)$	$\quad$ sampling ($x \sim D(y_1, \ldots, y_n)$)
$\quad \mathsf{Select}_n(x, v, y_1, \ldots, y_n)$	$\quad$ (de)multiplexing ($x = y_v$)
$\quad \mathsf{Gate}(x, G_1, G_2)$	$\quad$ gate (if x then G_1 else G_2)

In a graph $\mathsf{new}\ \overline{x : b}\ \mathsf{in}\ \{e_1, \ldots, e_m\}$, the variables x_i are bound; graphs are identified up to consistent renaming of bound variables. We write $\{e_1, \ldots, e_m\}$ for $\mathsf{new}\ \varepsilon\ \mathsf{in}\ \{e_1, \ldots, e_m\}$. We write $\mathrm{fv}(G)$ for the variables occurring free in G. If $x \notin \{\overline{y}\}$ we write $\mathsf{new}\ x : b'\ \mathsf{in}\ \mathsf{new}\ \overline{y : b}\ \mathsf{in}\ \{e_1, \ldots, e_m\}$ for the graph $\mathsf{new}\ x : b', \overline{y : b}\ \mathsf{in}\ \{e_1, \ldots, e_m\}$.

As a first example, the coin flipping code in Fun from Section 2.1 corresponds to the following factor graph:

Factor Graph for Coin Flipping Example:

$$
\begin{aligned}
G_{\mathrm{F}} = \{&\mathsf{Constant}_{0.5}(p),\\
&\mathsf{Sample}_{\mathbf{Bernoulli}}(x, p),\\
&\mathsf{Sample}_{\mathbf{Bernoulli}}(y, p),\\
&\mathsf{Binop}_{||}(z, x, y),\\
&\mathsf{Constant}_{\mathbf{true}}(z)\}
\end{aligned}
$$

For a second example, we give a factor graph G_{E} corresponding to the epidemology example of Section 2.4 (where $B = \mathbf{Bernoulli}$).

Factor Graph for Epidemiology Example:

$$
\begin{aligned}
G_{\mathrm{E}} = \{&\mathsf{Constant}_{0.01}(p_d), \mathsf{Sample}_B(\mathsf{has\_disease}, p_d),\\
&\mathsf{Gate}(\mathsf{has\_disease},\\
&\quad \mathsf{new}\ p_p : \mathbf{real}\ \mathsf{in}\ \{\mathsf{Constant}_{0.8}(p_p), \mathsf{Sample}_B(\mathsf{positive\_result}, p_p)\},\\
&\quad \mathsf{new}\ p_n : \mathbf{real}\ \mathsf{in}\ \{\mathsf{Constant}_{0.096}(p_n), \mathsf{Sample}_B(\mathsf{positive\_result}, p_n)\}),\\
&\mathsf{Constant}_{\mathbf{true}}(\mathsf{positive\_result})\}
\end{aligned}
$$

A factor graph typically denotes a probability distribution. The probability (density) of an assignment of values to variables is equal to the product of all the factors, averaged over all assignments to local variables. Here, we give a slightly more general semantics of factor graphs as measure transformers; the input measure corresponds to a prior factor over all variables that it mentions. Below, we use the Iverson brackets, where $[p]$ is 1 when p is true and 0 otherwise. We let $\delta(x = y) \triangleq \delta_0(x - y)$ when x, y denote real numbers, and $[x = y]$ otherwise.

Semantics of Factor Graphs: $\mathcal{J}[\![G]\!]^{\Sigma'}_{\Sigma} \in \mathtt{S}\langle\Sigma\rangle \rightsquigarrow \mathtt{S}\langle\Sigma,\Sigma'\rangle$

$$\mathcal{J}[\![G]\!]^{\Sigma'}_{\Sigma}\ \mu\ A \triangleq \int_A (\mathcal{J}[\![G]\!]\ s)\ d(\mu\times\lambda)(s)$$

$$\mathcal{J}[\![\mathsf{new}\ \overline{x:b}\ \mathsf{in}\ \{\overline{e}\}]\!]\ s \triangleq \int_{\mathbf{V}_{*_i b_i}} \prod_j (\mathcal{J}[\![e_j]\!]\ (s,\overline{x}))\, d\lambda(\overline{x})$$

$$\mathcal{J}[\![\mathsf{Equal}(l,l')]\!]\ s \triangleq \delta(\mathtt{lookup}\ l\ s = \mathtt{lookup}\ l'\ s)$$
$$\mathcal{J}[\![\mathsf{Constant}_c(l)]\!]\ s \triangleq \delta(\mathtt{lookup}\ l\ s = c)$$
$$\mathcal{J}[\![\mathsf{Binop}_{\otimes}(l,w_1,w_2)]\!]\ s \triangleq \delta(\mathtt{lookup}\ l\ s = \mathtt{lookup}\ w_1\ s \otimes \mathtt{lookup}\ w_2\ s)$$
$$\mathcal{J}[\![\mathsf{Sample}_D(l,v_1,\ldots,v_n)]\!]\ s \triangleq \mu_{D(\mathtt{lookup}\ v_1\ s,\ldots,\mathtt{lookup}\ v_n\ s)}(\mathtt{lookup}\ l\ s)$$
$$\mathcal{J}[\![\mathsf{Select}_n(l,v,y_1,\ldots,y_n)]\!]\ s \triangleq \prod_i \delta(l=y_i)^{[v=i]}$$
$$\mathcal{J}[\![\mathsf{Gate}(v,G_1,G_2)]\!]\ s \triangleq (\mathcal{J}[\![G_1]\!]\ s)^{[\mathtt{lookup}\ v\ s]} (\mathcal{J}[\![G_2]\!]\ s)^{[\neg\mathtt{lookup}\ v\ s]}$$

4.5 Factor Graph Semantics for Imp

An Imp statement has a straightforward semantics as a factor graph. Here, observation is defined by the value of the variable being the constant 0_b. We require that all local variables are declared at top-level, i.e. that in a composite statement $C_1;C_2$, neither C_1 nor C_2 are of the form **local** $l:b$ **in** C. Every Imp program can be rewritten in this form by pulling local variable declarations to the front. Moreover, if $\rho \vdash M \Rightarrow C,p$ then all local variables of C are declared at top-level.

Factor Graph Semantics of Imp: $G = \mathcal{G}[\![C]\!]$ **and** $\{e_1,\ldots e_n\} = \mathcal{E}[\![C]\!]$

$$\mathcal{G}[\![\textbf{local}\ l:b\ \textbf{in}\ C]\!] \triangleq \mathsf{new}\ l:b\ \mathsf{in}\ \mathcal{G}[\![C]\!]$$
$$\mathcal{G}[\![C]\!] \triangleq \mathsf{new}\ \varepsilon\ \mathsf{in}\ \mathcal{E}[\![C]\!] \qquad \text{if } C \neq \textbf{local}\ l:b\ \textbf{in} C'.$$

$$\mathcal{E}[\![\textbf{nil}]\!] \triangleq \varnothing$$
$$\mathcal{E}[\![C_1;C_2]\!] \triangleq \mathcal{E}[\![C_1]\!] \cup \mathcal{E}[\![C_2]\!]$$

$$\mathcal{E}[\![l \leftarrow c]\!] \triangleq \{\mathsf{Constant}_c(l)\}$$
$$\mathcal{E}[\![l \leftarrow l']\!] \triangleq \{\mathsf{Equal}(l,l')\}$$
$$\mathcal{E}[\![l \leftarrow l_1 \otimes l_2]\!] \triangleq \{\mathsf{Binop}_{\otimes}(l,l_1,l_2)\}$$
$$\mathcal{E}[\![l \overset{s}{\leftarrow} D(l_1,\ldots,l_n)]\!] \triangleq \{\mathsf{Sample}_D(l,l_1,\ldots,l_n)\}$$
$$\mathcal{E}[\![\textbf{observe}_b\ l]\!] \triangleq \{\mathsf{Constant}_{0_b}(l)\}$$
$$\mathcal{E}[\![\textbf{if}\ l\ \textbf{then}\ C_1\ \textbf{else}\ C_2]\!] \triangleq \{\mathsf{Gate}(l,\mathcal{G}[\![C_1]\!],\mathcal{G}[\![C_2]\!])\}$$

Factor Graph Semantics of Derived Forms:

$$\mathcal{G}[\![\textbf{local}\ \Sigma\ \textbf{in}\ C]\!] \triangleq \mathsf{new}\ \Sigma\ \mathsf{in}\ \mathcal{G}[\![C]\!]$$
$$\mathcal{G}[\![\textbf{if}\ l\ \textbf{then}_{\Sigma_1}\ C_1\ \textbf{else}_{\Sigma_2}\ C_2]\!] \triangleq \{\mathsf{Gate}(l,\mathsf{new}\ \Sigma_1\ \mathsf{in}\ \mathcal{G}[\![C_1]\!],\mathsf{new}\ \Sigma_2\ \mathsf{in}\ \mathcal{G}[\![C_2]\!])\}$$

The following theorem asserts that the two semantics for Imp coincide for compatible measures, which are defined as follows. If $T : t \rightsquigarrow u$ is a measure transformer composed from the combinators of Section 3 and $\mu \in \mathtt{M}\,t$, we say that T is *compatible* with μ if every application of `observe` f to some μ' in the evaluation of $T(\mu)$ satisfies either that f is discrete or that μ has a continuous density on some ε-neighbourhood of $f^{-1}(0.0)$.

This restriction is needed to ensure that the probabilistic semantics of the factor graph is well-defined.

Theorem 3. *If* $\Sigma \vdash C : \Sigma'$ *and* $\mu \in \mathtt{M}\langle \mathtt{S}\langle\Sigma\rangle\rangle$ *is compatible with* $\mathcal{I}[\![C]\!]$ *then* $\mathcal{I}[\![C]\!]\ \mu = \mathcal{J}[\![\mathcal{G}[\![C]\!]]\!]^{\Sigma'}_{\Sigma}\ \mu$.

Proof. By induction on the typing of C (Appendix A.3). □

5 Adding Arrays and Comprehensions

To be useful for machine learning, our language must support large datasets. To this end, we extend Fun and Imp with arrays and comprehensions. We offer three examples, after which we present the formal semantics, which is based on unrolling.

5.1 Comprehension Examples in Fun

Earlier, we tried to estimate the skill levels of three competitors in head-to-head games. Using comprehensions, we can model skill levels for an arbitrary number of players and games:

TrueSkill:

```
let trueskill (players:int[]) (results:(bool*int*int)[]) =
    let skills = [for p in players → random (Gaussian(10.0,20.0))]
    for (w,p1,p2) in results do
        let perf1 = random (Gaussian(skills.[p1], 1.0))
        let perf2 = random (Gaussian(skills.[p2], 1.0))
        if w // win?
        then observe (perf1 > perf2) // first player won
        else observe (perf1 = perf2) // draw
    skills
```

First, we create a prior distribution for each player: we assume that skills are normally distributed around 10.0, with variance 20.0. Then we look at each of the results—this is the comprehension. The result of the head-to-head matches is an array of triples: a Boolean and two indexes. If the Boolean is true, then the first index represents the winner and the second represents the loser. If the Boolean is false, then the match was a draw between the two players. The probabilistic program walks over the results, and observes that either the first player's performance—normally distributed around their skill level—was greater than the second's performance, or that the two players' performances were equal. Returning skills after these observations allows us to inspect the posterior distributions. Our original example can be modelled with players $= [0;1;2]$ (IDs for Alice, Bob, and Cyd, respectively) and results $= [(\mathbf{true},0,1);(\mathbf{true},1,2);(\mathbf{true},0,2)]$.

As another example, we can generalize the simple Bayesian classifier of Section 3 to arrays of categories and measurements, as follows:

Bayesian Inference Over Arrays:

```
let trainF (catIds:int[]) (trainData:(int*real)[]) fMean fVariance =
    let priors = [for cid in catIds → random (Gaussian(fMean,fVariance))]
    for (cid,m) in trainData do observe (m − random (Gaussian(priors.[cid],1.0)))
    priors
let catIds:int[] = (* ... *)
let trainingData:(int*real)[] = (* ... *)
```

The function trainF is a probabilistic program for training a naive Bayesian classifier on a single feature. Each category of objects—modelled by the array catIds—is given a normally distributed prior on the weight of objects in that category; we store these in the priors array. Then, for each measurement m of some object of category cid in the trainingData array, we observe that m is normally distributed according to the prior for that category of object. We then return the posterior distributions, which have been appropriately modified by the observed weights. We can train using this model by running a command such as classify catIds trainingData 20.0 5.0.

As a third example, consider the adPredictor component of the Bing search engine, which estimates the click-through rates for particular users on advertisements [11]. We describe a probabilistic program that models (a small part of) adPredictor. Without loss of generality, we use only two features to make our prediction: the advertiser's listing and the phrase used for searching. In the real system, many more (undisclosed) features are used for prediction.

The following is an F# fragment of our adPredictor model.

adPredictor in F#:

```
let read_lines filename count line = (* ... *)
[<RegisterArray>]
let imps = (* ... *)
[<ReflectedDefinition>]
let probit b x =
    let y = random (Gaussian(x,1.0))
    observe (b == (y > 0.0))
[<ReflectedDefinition>]
let ad_predictor (listings:int[]) (phrases:int[]) impressions =
    let lws = [for l in listings → random (Gaussian(0.0,0.33))]
    let pws = [for p in phrases → random (Gaussian(0.0,0.33))]
    for (clicked,lid,pid) in Array.toList impressions do
        probit clicked (lws.[lid] + pws.[pid])
    lws,pws(lws.[lid]
```

The read_lines function loads data from a file on disk. The data are formatted as newline-separated records of comma-separated values. There are three important values in each record: a field that is 1 if the given impression lead to a click, and a 0 otherwise; a field

that is the database ID of the listing shown; a field that is the part of the search phrase that led to the selection of the listing. We preprocess the data in three ways, which are elided in the code above. First, we convert the 1/0-valued Boolean to a **true/false**-valued Boolean. Second, we normalize the listing IDs so that they begin at 0, that is, so that we can use them as array indexes. Third, we collect unique phrases and assign them fresh, 0-based IDs. We define imps—a list of advertising impressions (a listing ID and a phrase ID) and whether or not the ad was clicked—in terms of this processed data. The [<RegisterArray>] attribute on the definition of imps instructs the compiler to simply evaluate this F# expression, yielding a deterministic constant. Finally, ad_predictor defines the model. We use the [<ReflectedDefinition>] attribute on ad_predictor to mark it as a probabilistic program, which should be compiled and sent to Infer.NET. Suppose we have stored the collated listing and phrase IDs in ls and ps, respectively; we can train on the impressions by calling ad_predictor ls ps imps.

5.2 Formalizing Arrays and Comprehensions in Fun

We introduce syntax for arrays in Fun, and give interpretations of this extended syntax in terms of the core languages, essentially by treating arrays as tuples and by unfolding iterations. We work with non-empty zero-indexed arrays of statically known size (representing, for example, statically known experimental data).

There are three array operations: array literals, indexing, and array comprehension. First, let $\mathcal{R}$ be a set of *ranges* r. Ranges allow us to differentiate arrays of different sizes. Moreover, limitations in our underlying factor-graph implementation disallow nested iterations on the same range. Here we disallow nested iterations altogether—they are not needed for our examples and they would significantly complicate the formalism. We assign sizes to ranges using the function $|\cdot| : \mathcal{R} \to \mathbb{Z}^+$. In the metalanguage, arrays over range r correspond to tuples of length $|r|$.

Extended Syntax of Fun:

$t ::= \cdots \mid t[r]$	type
$M, N ::= \cdots \mid$	expression
$\quad [V_1; \ldots; V_n]$	array literal
$\quad V_1.[V_2]_r$	indexing
$\quad [\textbf{for } x \textbf{ in}_r V \to M]$	comprehension

First, we add arrays as a type: $t[r]$ is an array of elements of type t over the range r. In the array type $t[r]$, we require that the type t contains no array type $t'[r']$, that is, we do not consider nested arrays. Indexing, $V_1.[V_2]_r$, extracts elements out of an array, where the index V_2 is computed modulo the size $|r|$ of the array V_1. A comprehension $[\textbf{for } x \textbf{ in}_r V \to M]$ maps over an array V, producing a new array where each element is determined by evaluating M with the corresponding element of array V bound to x. To simplify the formalism, we here require that the body M of the comprehension contains neither array literals nor comprehensions. We attach the range to indexing and comprehensions so that the measure transformer semantics can be given simply; the range can be inferred easily, and need not be written by the programmer. We elide the range in our code examples.

In this formalism, we do not distinguish comprehensions that produce values—like the one that produces skills—and those that do not—like the one that observes player performances according to results. For the sake of efficiency, our implementation does distinguish these two uses. In some of the code examples, we write **for** x **in** V **do** M to mean [**for** x **in**$_r$ $V \to M$]. We do so only when M has type **unit** and we intend to ignore the result of the expression.

We encode arrays as tuples. For all $n > 0$, we define $\pi_n(M,N)$ with $M : t^n$ and $N :$ **int** and if $N\%n = i$ we expect $\pi_n((V_0,\ldots,V_{n-1}),N) = V_i$.

Derived Types and Expressions for Arrays in Fun:

$\pi_1(M,N) := M$
$\pi_n(M,N) :=$ **if** $N\%n == 0$ **then** $M.1$ **else** $\pi_{n-1}(M.2,N-1)$ for $n > 1$

$t[r] := t^{|r|}$ where $t^1 := t$ and $t^{n+1} := t * t^n$

$[V_0;\ldots;V_{n-1}] := (V_0,\ldots,V_{n-1})$
$V_1[V_2]_r := \pi_{|r|}(V_1,V_2)$
for x **in**$_r$ $V \to M :=$
  let $y_0 =$ (**let** $x = \pi_{|r|}(V,0)$ **in** M) **in**
  $\ldots$
  let $y_{|r|-1} =$ (**let** $x = \pi_{|r|}(V,|r|-1)$ **in** M) **in**
  $(y_0;\ldots;y_{|r|-1})$ where $y_1,\ldots,y_{|r|}$ are fresh for M and V.

Our derived forms for arrays yield programs that scale with the data over which they compute—we implement $V[i]_r$ with $O(|r|)$ projections. To avoid this problem, our implementation takes advantage of support for arrays in the Infer.NET factor graph library (see Section 5.3).

The static semantics of these new constructs is straightforward; we give the derived rules for (FUN ARRAY), (FUN INDEX), and (FUN FOR). By adding these as derived forms in Fun, we do not need to extend Imp at all. On the other hand, our formalism does not reflect that our implementation preserves the structure of array comprehensions when going to Infer.NET.

Extended Typing Rules for Fun Expressions: $\Gamma \vdash M : t$

(FUN ARRAY)
$$\frac{\Gamma \vdash V_i : t \quad \forall i \in 0..n-1}{\Gamma \vdash [V_0;\ldots;V_{n-1}] : t[r_n]}$$

(FUN INDEX)
$$\frac{\Gamma \vdash V_1 : t[r] \quad \Gamma \vdash V_2 : \textbf{int}}{\Gamma \vdash V_1[V_2]_r : t}$$

(FUN FOR)
$$\frac{\Gamma \vdash V : t[r] \quad \Gamma, x : t \vdash M : t'}{\Gamma \vdash [\textbf{for } x \textbf{ in}_r V \to M] : t'[r]}$$

The rule (FUN ARRAY) uses the notation r_n for the *concrete range* of size n; we assume there is a unique such range for each $n > 0$. This rule can be derived using repeated applications of (FUN PAIR). The rule (FUN INDEX) asks for $V_1[V_2]_r$ to be well-typed that V_1 is a non-empty array and V_2 is an integer; the actual index is the value of V_2 modulo the size of the array as in the meta-language. We can derive this rule for a given n by induction on n, using repeated applications of (FUN IF); we use (FUN PROJ1) in the **then** case and (FUN PROJ2) in the **else** case. The rule (FUN FOR)

asks for $[\textbf{for } x \textbf{ in}_r\ V \to M]$ to be well-typed that the source expression V be an array, and that the yielding expression M be well-typed assuming a suitable type for x. We can derive (FUN FOR) using repeated applications of (FUN LET), with (FUN PAIR) to type the final result.

5.3 Arrays in Imp

We now sketch our structure-preserving implementation strategy. We work in a version of Imp with arrays and iteration over ranges, and we extend both the assignment form and expressions to permit array indexing. Inside the body of an iteration over a range, the name of the range can be used as an index.

Extended Syntax of Imp:

$E ::= \ldots \mid l[l'] \mid l[r]$	expression
$I ::= \cdots \mid$	statement
$\quad l[r] \leftarrow E$	$\quad$ assignment to array item
$\quad \textbf{for } r \textbf{ do } C$	$\quad$ iteration over ranges

We require that every occurrence of an index r is inside an iteration $\textbf{for } r \textbf{ do } C$. Inside such an iteration, every assignment to an array variable must be at index r. We also extend patterns to include range indexed locations, and write $(p_1, p_2)[r]$ for $(p_1[r], p_2[r])$.

Our compiler translates comprehensions over variables of array type as an iteration over the translation of the body of the comprehension. We add to ρ the fact that the comprehension variable corresponds to the array variable indexed by the range. We invent a fresh array result pattern p', and assign the result of the translated body to $p'[r]$. Finally, we hide the local variables of the translation of the body of the comprehension, in order to avoid clashes in the unrolling semantics of the loop. This compilation corresponds to the rule (TRANS FOR) below. In particular, the sizes of ranges are never needed in our compiler, so compilation is not data dependent.

Compilation of comprehensions:

(TRANS FOR)

$$\frac{\rho\{x \mapsto \rho(z)[r]\} \vdash M \Rightarrow C, p \qquad p[r] \sim p' \qquad (\mathrm{locs}(\rho) \cup \mathrm{locs}(C)) \cap \mathrm{fv}(p') = \varnothing}{\rho \vdash [\textbf{for } x \textbf{ in}_r\ z \to M] \Rightarrow \textbf{for } r \textbf{ do local } \mathrm{locs}(C) \textbf{ in } (C; p'[r] \leftarrow p), p'}$$

6 Implementation Experience

We implemented a compiler from Fun to Imp in F#. We wrote two backends for Imp: an exact inference algorithm based on a direct implementation of measure transformers for discrete measures, and an approximating inference algorithm for continuous measures, using Infer.NET [30]. The translation of Section 4 formalizes our translation of Fun to Imp. Translating Imp to Infer.NET is relatively straightforward, and amounts to a syntax-directed series of calls to Infer.NET's object-oriented API.

The frontend of our compiler takes (a subset of) actual F# code as its input. To do so, we make use of F#'s *reflected definitions*, which allow programmatic access to ASTs. This implementation strategy is advantageous in several ways. First, there is no need to design new syntax, or even write a parser. Second, all inputs to our compiler are typed ASTs of well typed F# programs. Third, a single file can contain both ordinary F# code as well as reflected definitions. This allows a single module to both read and process data, and to specify a probabilistic model for inference from the data.

Functions computing array values containing deterministic data are tagged with an attribute RegisterArray, to signal to the compiler that they do not need to be interpreted as Fun programs. Reflected definitions later in the same file are typed with respect to these registered definitions and then run in Infer.NET with the pre-processed data; we discuss this idea more below.

Below follows some statistics on a few of the examples we have implemented. The number of lines of code includes F# code that loads and processes data from disk before loading it into Infer.NET. The times are based on an average of three runs. All of the runs are on a four-core machine with 4GB of RAM. The Naive Bayes program is the naive Bayesian classifier of the earlier examples. The Mixture model is another clustering/classification model. TrueSkill and adPredictor were described earlier. TrueSkill spends the majority of its time (64%) in Infer.NET, performing inference. AdPredictor spends most of the time in pre-processing (58%), and only 40% in inference. The time spent in our compiler is negligible, never more than a few hundred milliseconds.

Summary of our Basic Test Suite:

	LOC	Observations	Variables	Time
Naive Bayes	28	9	3	<1s
Mixture	33	3	3	<1s
TrueSkill	68	15,664	84	6s
adPredictor	78	300,752	299,594	3m30s

In summary, our implementation strategy allowed us to build an effective prototype quickly and easily: the entire compiler is only 2079 lines of F#; the Infer.NET backend is 600 lines; the discrete backend is 252 lines. Our implementation, however, is only a prototype, and has limitations. Our discrete backend is limited to small models using only finite measures. Infer.NET supports only a limited set of operations on specific combinations of probabilistic and deterministic arguments. It would be useful in the future to have an enhanced type system able to detect errors arising from illegal combinations of operators in Infer.NET. The reflected definition facility is somewhat limited in F#. In the example below, a call to Array.toList is required because F# does not reflect definitions that contain comprehensions over arrays—only lists. (The F# to Fun compiler discards this extra call as a no-op, so there is no runtime overhead.)

7 Related Work

To the best of our knowledge, this paper introduces the first rigorous measure-theoretic semantics shown to be in agreement with a factor graph semantics for a probabilistic

language with observation and sampling from continuous distributions. Hence, we lay a firm foundation for inference on probabilistic programs via modern message-passing algorithms on factor graphs.

Formal Semantics of Probabilistic Languages There is a long history of formal semantics for probabilistic languages with sampling primitives, often combined with recursive computation. One of the first semantics is for Probabilistic LCF [42], which augments the core functional language LCF with weighted binary choice, for discrete distributions. (Apart from its inclusion of observations, Bernoulli Fun is a first-order terminating form of Probabilistic LCF.) Kozen [21] develops a probabilistic semantics for while-programs augmented with random assignment. He develops two provably equivalent semantics; one more operational, and the other a denotational semantics using partially ordered Banach spaces. Imp is simpler than Kozen's language, as Imp has no unbounded while-statements, so the semantics of Imp need not deal with nontermination. On the other hand, observations are not present in Kozen's language.

Jones and Plotkin [16] investigate the probability monad, and apply it to languages with discrete probabilistic choice. Ramsey and Pfeffer [39] give a stochastic λ-calculus with a measure-theoretic semantics in the probability monad, and provide an embedding within Haskell; they do not consider observations. We can generalize the semantics of **observe** to the stochastic λ-calculus as filtering in the probability monad (yielding what we may call a sub-probability monad), as long as the events that are being observed are discrete or have non-zero probability. In their notation, we can augment their language with a failure construct defined by $\mathcal{P}[\![\textbf{fail}]\!]\rho = \mu_0$ where we define $\mu_0(A) = 0$ for all measurable sets A. Then, we can define **observe** $v = (\textbf{if } v = \textbf{true then } () \textbf{ else fail})$. However, as discussed in Section 3.5, zero-probability observations of real variables do not translate easily to the probability monad, as the following example shows. Let N be an expression returning a continuous distribution, for example, **sample** (**Gaussian** (0.0,1.0)), and let f x = **observe** x. Suppose there is a semantics for $[\![\mathsf{f}\ \mathsf{x}]\!]\{\mathsf{x} \mapsto r\}$ for real r in the probability monad. The probability monad semantics of the program **let** x = N **in** f x of the stochastic λ-calculus is $[\![N]\!] \gg\!= \lambda y.[\![\mathsf{f}\ \mathsf{x}]\!]\{\mathsf{x} \mapsto y\}$, which yields the measure $\mu(A) = \int_{\mathbb{R}} (\mathbf{M}[\![[\![\mathsf{f}\ \mathsf{x}]\!]\{\mathsf{x} \mapsto y\}]\!])(A)\, d\mathbf{M}[N](y)$. Here the probability $(\mathbf{M}[\![[\![\mathsf{f}\ \mathsf{x}]\!]\{x \mapsto y\}]\!])(A)$ is zero except when $y = 0$, where it is some real number. Since the N-measure of $y = 0$ is zero, the whole integral is zero for all A (in particular $\mu(\mathbb{R}) = 0$), whereas the intended semantics is that x is constrained to be zero with probability 1 (so in particular $\mu(\mathbb{R}) = 1$).

The probabilistic concurrent constraint programming language pcc of Gupta, Jagadeesan, and Panangaden [12] is also intended for describing probability distributions using independent sampling and constraints. Our use of observations corresponds to constraints on random variables in pcc. In the finite case, pcc also relies on a sampling semantics with observation (constraints) denoting filtering. To admit continuous distributions, pcc adds general fixpoints and defines the semantics of a program as the limit of finite unrollings of its fixpoints, if defined. This can lead to surprising results, such as that the distribution resulting from observing that two uniform distributions are equal may not itself be uniform. In contrast, our goal is an efficient implementation via

factor graphs, which led us to work directly with standard distributions and to have a semantics of observation that is independent of the program text.

McIver and Morgan [27] develop a theory of abstraction and refinement for probabilistic while programs, based on weakest preconditions. They reject a subdistribution transformer semantics in order to admit demonic nondeterminism in the language.

We conjecture that Fun and Imp could in principle be conferred semantics within a probabilistic language supporting general recursion, by encoding observations by placing the whole program within a conditional sampling loop, and by encoding Gaussian and other continuous distributions as repeated sampling using recursive functions. Still, our choices in formulating the semantics of Fun and Imp were to include some distributions as primitive, and to exclude recursion; compared to encodings within probabilistic languages with recursion, these choices have some advantages: (1) our measure transformer semantics relies on relatively elementary measure theory, with no need to express non-termination or to compute limits when defining the model; (2) our semantics is compositional rather than relying on a global sampling loop; and (3) our semantics has a direct implementation via message-passing algorithms on factor graphs, with efficient implementations of primitive distributions.

Recent work on semantics of probabilistic programs within interactive theorem provers includes the mechanization of measure theory [14] and Lebesgue integration [29] in HOL, and proofs of randomized algorithms in Coq [2].

Probabilistic Languages for Machine Learning Koller et al. [20] pioneered the idea of representing a probability distribution using first-order functional programs with discrete random choice, and proposed an inference algorithm for Bayesian networks and stochastic context-free grammars. Observations happen outside their language, by returning the distributions $\mathrm{P}[A \wedge B], \mathrm{P}[A \wedge \neg B], \mathrm{P}[\neg A]$ which can be used to compute $\mathrm{P}[B \mid A]$.

Park et al. [35] propose $\lambda_\circ$, the first probabilistic language with formal semantics applied to actual machine learning problems involving continuous distributions. The formal basis is sampling functions, which uniformly supports both discrete and continuous probability distributions, and inference is by Monte Carlo methods. The calculus $\lambda_\circ$ does not include observations, but enables conditional sampling via fixpoints and rejection.

Infer.NET [30] is a software library that implements the approximate deterministic algorithms expectation propagation [32] and variational message passing [45], as well as Gibbs sampling, a nondeterministic algorithm. Infer.NET models are written in a probabilistic subset of C#, known as Csoft [44]. Csoft allows **observe** on zero probability events, but its semantics has not previously been formalized and it is currently only implemented as an internal language of Infer.NET. IBAL [37] has observations and uses a factor graph semantics, but only works with discrete datatypes and thus does not need advanced probability theory. Moreover, there seems to be no proof that the factor graph denotation of an IBAL program yields the same distribution as the direct semantics, an important goal of the present work. HANSEI [18,17] is a programming library for OCaml, based on explicit manipulation of discrete probability distributions as lists, and sampling algorithms based on coroutines. HANSEI uses an explicit `fail`

statement, which is equivalent to **observe false** and so cannot be used for conditioning on zero probability events.

FACTORIE [26] is a Scala library for explicitly constructing factor graphs. Although there are many Bayesian modelling languages, Csoft and IBAL are the only previous languages implemented by a compilation to factor graphs. Probabilistic Scheme [38] is a probabilistic form of the untyped functional language Scheme, limited to discrete distributions, and with a construct for reifying the distribution induced by a thunk as a value. Church [10] is another probabilistic form of Scheme, equipped with conditional sampling and a mechanism of stochastic memoization. Queries are implemented using Monte Carlo methods. Blaise [5] supports the compositional construction of sophisticated probabilistic models, and decouples the choice of inference algorithm from the specification of the distribution. WinBUGS [33] is a popular language for explicitly describing distributions suitable for Monte Carlo analysis.

Other Uses of Probabilistic Languages Probabilistic languages with formal semantics find application in many areas apart from machine learning, including databases [7], model checking [23], differential privacy [28,40], information flow [24], and cryptography [1]. A recent monograph on semantics for labelled Markov processes [34] focuses on bisimulation-based equational reasoning. The syntax and semantics of Imp is modelled on the probabilistic language pWhile [3] without observations.

Erwig and Kollmansberger [8] describe a library for probabilistic functional programming in Haskell. The library is based on the probability monad, and uses a finite representation suitable for small discrete distributions; the library would not suffice to provide a semantics for Fun or Imp with their continuous and hybrid distributions. Their library has similar functionality to that provided by our combinators for discrete distributions listed in Appendix B.

8 Conclusion

We advocate probabilistic functional programming with observations and comprehensions as a modelling language for Bayesian reasoning. We developed a system based on the idea, invented new formal semantics to establish correctness, and evaluated the system on a series of typical inference problems.

Our direct contribution is a rigorous semantics for a probabilistic programming language that also has an equivalent factor graph semantics. We have shown that probabilistic functional programs with iteration over arrays, but without the complexities of general recursion, are a concise representation for complex probability distributions arising in machine learning. An implication of our work for the machine learning community is that probabilistic programs can be written directly within an existing declarative language (Fun—a subset of F#), linked by comprehensions to large datasets, and compiled down to lower level Bayesian inference engines.

For the programming language community, our new semantics suggests some novel directions for research. What other primitives are possible—non-generative models, inspection of distributions, on-line inference on data streams? Can we verify the transformations performed by machine learning compilers such as Infer.NET compiler for

Csoft? What is the role of type systems for such probabilistic languages? Avoiding zero probability exceptions, and ensuring that we only generate factor graphs suitable for our back-end, are two possibilities, but we expect there are more.

Acknowledgements We gratefully acknowledge discussions with and comments from Ralf Herbrich, Oleg Kiselyov, Tom Minka, Aditya Nori, Chung-chieh Shan, Robert Simmons, Nikhil Swamy, Dimitrios Vytiniotis, John Winn, and the anonymous reviewers.

A Detailed Proofs

Our proofs are structured as follows.

- Appendix A.1 gives a proof of Proposition 2.
- Appendix A.2 gives a proof of Theorem 2.
- Appendix A.3 gives a proof for Theorem 3.

A.1 Proof of Proposition 2

We begin with a series of lemmas.

Lemma 11 (Pattern agreement weakening). *If $\Sigma \vdash p : t$ and $\Sigma, \Sigma' \vdash \diamond$, then $\Sigma, \Sigma' \vdash p : t$.*

Proof. By induction on t. □

Lemma 12 (Expression and statement heap weakening).

(1) *If $\Sigma \vdash E : b$ and $\Sigma, \Sigma' \vdash \diamond$, then $\Sigma, \Sigma' \vdash E : b$*
(2) *If $\Sigma \vdash I : \Sigma'$ and $\Sigma, \Sigma', \Sigma'' \vdash \diamond$, then $\Sigma, \Sigma'' \vdash I : \Sigma'$*
(3) *If $\Sigma \vdash C : \Sigma'$ and $\Sigma, \Sigma', \Sigma'' \vdash \diamond$, then $\Sigma, \Sigma'' \vdash C : \Sigma'$.*

Proof. By induction on E, I, and C, respectively. □

Lemma 13 (Pattern agreement uniqueness). *If $\Sigma \vdash p : t$ and $\Sigma' \vdash p' : t$ then $p \sim p'$.*

Proof. By induction on t. □

Lemma 14 (Pattern creation). *If $\Sigma \vdash p : t$ then there exists Σ' such that $\Sigma, \Sigma' \vdash \diamond$ and $\Sigma' \vdash p' : t$ and $\mathrm{dom}(\Sigma') = \mathrm{fv}(p')$.*

Proof. By induction on t, and the assumption that there always exist new, globally fresh locations. □

Lemma 15 (Pattern assignment). *If $\Sigma \vdash p : t$ and $\Sigma' \vdash p' : t$ and $\Sigma, \Sigma' \vdash \diamond$, then $\Sigma \vdash p' \leftarrow p : \Sigma''$, where $\Sigma'' \subseteq \Sigma'$.*

Proof. By induction on t.

- ($t = \mathbf{unit}$) Trivial: $p' \leftarrow p = \mathbf{nil}$, so $\Sigma'' = \varepsilon \subseteq \Sigma'$.
- ($t = \mathbf{bool}$) $\Sigma \vdash l : \mathbf{bool}$ and $\Sigma' \vdash l' : \mathbf{bool}$, so $l : \mathbf{bool} \in \Sigma$ and $l' : \mathbf{bool} in \Sigma'$. So $l : \mathbf{bool} \vdash l' \leftarrow l : (l' : \mathbf{bool}) \subseteq \Sigma'$.
- ($t = \mathbf{int}$) Similar.
- ($t = \mathbf{real}$) Similar.
- ($t = t_1 * t_2$) $\Sigma \vdash p_1, p_2 : t_1 * t_2$ and $\Sigma' \vdash p'_1, p'_2 : t_1 * t_2$. Both Σ and Σ' factor into contexts that type p_1 and p_2 (resp. p'_1 and p'_2) individually; call them Σ_1 and Σ_2 (resp. Σ'_1 and Σ'_2). By the IHs, we have $\Sigma_1 \vdash p'_1 \leftarrow p_1 : \Sigma''_1 \subseteq \Sigma'_1 and \Sigma_2 \vdash p'_2 \leftarrow p_2 : \Sigma''_2 \subseteq \Sigma'_2$. We can then see $\Sigma \vdash p'_1 \leftarrow p_1; p'_2 \leftarrow p_2 : \Sigma''_1, \Sigma''_2 \subseteq \Sigma'_1, \Sigma'_2$.

□

The purpose of this subsection is to prove the following.

Restatement of Proposition 2 *Suppose $\Gamma \vdash M : t$ and $\Sigma \vdash \rho : \Gamma$.*

(1) *There are C and p such that $\rho \vdash M \Rightarrow C, p$.*
(2) *Whenever $\rho \vdash M \Rightarrow C, p$, there is Σ' such that $\Sigma \vdash C : \Sigma'$ and $\Sigma, \Sigma' \vdash p : t$.*

Proof. By induction on the typing of M, leaving Σ and ρ general.

(FUN VAR) $\Gamma \vdash x : t$. For (1), we have $C = \mathbf{nil}$ and $p = \rho(x)$. For (2), let $\Sigma' = \varepsilon$. By assumption, $\Sigma, \Sigma' \vdash \rho(x) : t$ and $\Sigma \vdash \mathbf{nil} : \Sigma'$ immediately.

(FUN CONST) $\Gamma \vdash c : ty(c)$. For (1), we have:

$$\begin{array}{l} l \notin \mathrm{locs}(\rho) \\ ty(c) = b \text{ for some base type b} \\ \rho \vdash c \Rightarrow l \leftarrow c, l \end{array}$$

For (2), let $\Sigma' = l : ty(c)$. We have $\Sigma, \Sigma' \vdash l : ty(c)$ and $S' \vdash l \leftarrow c : S'$.

(FUN OPERATOR) $\Gamma \vdash V_1 \otimes V_2 : b_3$, where $\otimes$ has takes $b_1 * b_2 \to b_3$. By inversion and the IH:

$$\begin{array}{ll} \Gamma \vdash V_1 : b_1 & \\ \rho \vdash V_1 \Rightarrow C_1, l_1 & (IH_1) \\ \exists \Sigma_1 & (IH_2) \\ \quad \Sigma, \Sigma_1 \vdash l_1 : b_1 & \\ \quad \Sigma \vdash C_1 : \Sigma_1 & \\ \Gamma \vdash V_2 : b_2 & \\ \rho \vdash V_2 \Rightarrow C_2, l_2 & (IH_2) \\ \exists \Sigma_2 & (IH_2) \\ \quad \Sigma, \Sigma_2 \vdash l_2 : b_2 & \\ \quad \Sigma \vdash C_2 : \Sigma_2 & \end{array}$$

We have for (1), by (TRANS OPERATOR): $\rho \vdash V_1 \otimes V_2 \Rightarrow C_1; C_2; l \leftarrow l_1 \otimes l_2, l$. Let $\Sigma' = \Sigma_1, \Sigma_2, l : b_3 \vdash \diamond$. By weakening we find for (2): $\Sigma, \Sigma' \vdash l : b_3$ and $\Sigma \vdash C_1; C_2; l \leftarrow l_1 \otimes l_2 : \Sigma'$.

(FUN PAIR) $\Gamma \vdash (M_1, M_2) : t_1 * t_2$. By inversion and the IH:

$$\begin{array}{ll} \Gamma \vdash M_1 : t_1 & \\ \rho \vdash M_1 \Rightarrow C_{M_1}, p_1 & (IH_1) \\ \exists \Sigma_1 & (IH_2) \\ \quad \Sigma, \Sigma_1 \vdash p_1 : t_1 & \\ \quad \Sigma \vdash C_{M_1} : \Sigma_1 & \\ \Gamma \vdash M_2 : t_2 & \\ \rho \vdash M_2 \Rightarrow C_{M_2}, p_2 & (IH_1) \\ \exists \Sigma_2 & (IH_2) \\ \quad \Sigma, \Sigma_2 \vdash p_2 : t_2 & \\ \quad \Sigma \vdash C_{M_2} : \Sigma_2 & \end{array}$$

We have for (1): $\rho \vdash (M_1, M_2) \Rightarrow C_{M_1}; C_{M_2}, (p_1, p_2)$. Let $\Sigma' = \Sigma_1, \Sigma_2 \vdash \diamond$. By weakening we find for (2): $\Sigma, \Sigma' \vdash (p_1, p_2) : t_1 * t_2$ and $\Sigma \vdash C_{M_1}; C_{M_2} : \Sigma'$.

(FUN PROJ1) $\Gamma \vdash M.1 : t_1$. By inversion and the IH:

$$\begin{array}{ll} \Gamma \vdash M : t_1 * t_2 & \\ \rho \vdash M \Rightarrow C_M, p & (IH_1) \\ \exists \Sigma' & (IH_2) \\ \quad \Sigma, \Sigma' \vdash p : t_1 * t_2 & \\ \quad \Sigma \vdash M : S' & \end{array}$$

By inversion, $p = (p_1, p_2)$, such that $\Sigma, \Sigma' \vdash p_1 : t_1$ and $\Sigma, \Sigma' \vdash p_2 : t_2$. We now have $\rho \vdash M.1 \Rightarrow C_M, p_1$ for (1). We use Σ' to show $\Sigma, \Sigma' \vdash p_1 : t_1$ and $\Sigma \vdash C_M : \Sigma'$ for (2).

(FUN PROJ2) $\Gamma \vdash M.2 : t_2$. Analogous to the previous case.

(FUN IF) $\Gamma \vdash \textbf{if } M_1 \textbf{ then } M_2 \textbf{ else } M3 : t$. We have:

$$\begin{array}{ll} \Gamma \vdash M_1 : \textbf{bool} & \\ \rho \vdash M_1 \Rightarrow C_{M_1}, p_1 & (IH_1) \\ \exists \Sigma_1 & (IH_2) \\ \quad \Sigma, \Sigma_1 \vdash p_1 : \textbf{bool} & \\ \quad \Sigma \vdash C_{M_1} : \Sigma_1 & \\ \Gamma \vdash M_2 : t & \\ \rho\{x \mapsto p_l\} \vdash M_2 \Rightarrow C_{M_2}, p_2 & (IH_1) \\ \exists \Sigma_2 & (IH_2) \\ \quad \Sigma, \Sigma_2 \vdash p_2 : t & \\ \quad \Sigma \vdash C_{M_2} : \Sigma_2 & \\ \Gamma \vdash M_3 : t & \\ \rho\{x \mapsto p_r\} \vdash M_3 \Rightarrow C_{M_3}, p_3 & (IH_1) \\ \exists \Sigma_3 & (IH_2) \\ \quad \Sigma, \Sigma_3 \vdash p_3 : t & \\ \quad \Sigma \vdash C_{M_3} : \Sigma_3 & \end{array}$$

By inversion, $p_1 = l$ and $\Sigma, \Sigma_1 \vdash l : \textbf{bool}$. By pattern agreement uniqueness (Lemma 13), $p_2 \sim p_3$. Let $\Sigma_{p'} \vdash p' : t$, for $\mathrm{dom}(\Sigma_{p'}) = fv(p)$ (by Lemma 14). We have $(\mathrm{locs}(\rho) \cup \mathrm{locs}(C_1) \cup \mathrm{locs}(C_2) \cup \mathrm{locs}(C_3)) \cap fv(p) = \varnothing$. We also have $p' \sim p_2$ and $p' \sim p_3$. We now have for (1):

$$\begin{array}{l} \rho \vdash \textbf{if } M_1 \textbf{ then } M_2 \textbf{ else } M_3 \Rightarrow \\ C_{M_1}; \textbf{if } l \textbf{ then local } \mathrm{locs}(C_2) \textbf{ in } C_{M_2}; [[p' \leftarrow p_2]] \textbf{ else local } \mathrm{locs}(C_3) \textbf{ in } C_{M_3}; [[p' \leftarrow p_3]], p' \end{array}$$

Finally, let $\Sigma_f = \Sigma_2 \cap \Sigma_3 \cap \Sigma_{p'} \vdash \diamond$ and $\Sigma' = \Sigma_1, \Sigma_f \vdash \diamond$. By pattern assignment, we can see $\Sigma_f \vdash [[p' \leftarrow p_2]]$ and $\Sigma_f \vdash [[p' \leftarrow p_3]]$. By weakening (Lemmas 11, and 12) we have what we need for (2):

$$\begin{array}{l} \Sigma, \Sigma' \vdash p' : t \\ \Sigma \vdash C_{M_1}; \textbf{if } l \textbf{ then } \ldots \textbf{ else } \ldots : \Sigma' \end{array}$$

(FUN LET) $\Gamma \vdash \textbf{let}\ x = M_1\ \textbf{in}\ M_2 : t_2$. We have:

$$\begin{array}{ll} \Gamma \vdash M_1 : t_1 & \\ \rho \vdash M_1 \Rightarrow C_{M_1}, p & (IH_1) \\ \exists \Sigma_1 & (IH_2) \\ \quad \Sigma, \Sigma_1 \vdash p_1 : t_1 & \\ \quad \Sigma \vdash C_{M_1} : \Sigma_1 & \\ \Gamma, x : T_1 \vdash M_2 : t_2 & \end{array}$$

Next, note that $\Sigma, \Sigma_1 \vdash \rho\{x \mapsto p_1\} : \Gamma, x : T_1$. We can now apply the IH to M_2's typing derivation to see:

$$\begin{array}{ll} \rho\{x \mapsto p_1\} \vdash M_2 \Rightarrow C_{M_2}, p_2 & (IH_1) \\ \exists \Sigma_2 & (IH_2) \\ \quad \Sigma, \Sigma_2 \vdash p_2 : t_2 & \\ \quad \Sigma \vdash C_{M_2} : \Sigma_2 & \end{array}$$

First, we have: $\rho \vdash \textbf{let}\ x = M_1\ \textbf{in}\ M_2 \Rightarrow C_{M_1}; C_{M_2}, p_2$ for (1). For (2), let $\Sigma' = \Sigma_1, \Sigma_2 \vdash \diamond$. By weakening, we find $\Sigma, \Sigma' \vdash p_2 : t_2$ and $\Sigma \vdash C_{M_1}; C_{M_2} : \Sigma'$.

(FUN OBSERVE) $\Gamma \vdash \textbf{observe}_b\ E : \textbf{unit}$. By the IH, with $\Sigma' = \varepsilon$ from IH_2.

(FUN RANDOM) $\Gamma \vdash \textbf{random}(D(V)) : b_{n+1}$. We have:

$$\begin{array}{l} D : (x_1 : b_1 * \ldots * x_n : b_n) \to b_{n+1} \\ \Gamma \vdash V : (b_1 * \ldots * b_n) \end{array}$$

We have, by the IH:

$$\begin{array}{ll} \rho \vdash V \Rightarrow C, p & (IH_1) \\ \exists \Sigma' & (IH_2) \\ \quad \Sigma, \Sigma' \vdash p : t & (*) \\ \quad \Sigma \vdash C : \Sigma' & \end{array}$$

So $\rho \vdash \textbf{random}(D(V)) \Rightarrow C; l \xleftarrow{s} D(p), l$, for (1). We find (2) by (*) and by (Imp Seq), (Imp Random), and the IH $\Sigma \vdash C; l : \Sigma', l$, where $\Sigma', l \vdash l : b_{n+1}$. □

A.2 Proof of Theorem 2

We use the following lemma.

Lemma 16 (Value equivalence). *If $\Gamma \vdash V : t$ and $\Sigma \vdash \rho : \Gamma$ and $\rho \vdash V \Rightarrow C, p$ then $\mathcal{A}[\![C]\!] = \texttt{pure}\ f$, where f is either id or a series of (independent) calls to* `add` *:*

$$f = \lambda s.\ \texttt{add}\ l_1(\texttt{add}\ l_2(\ldots(\texttt{add}\ l_n(s, c_n))\ldots, c_2), c_1)$$

where each of the l_i are distinct, and

$$\mathcal{A}[\![V]\!] = \texttt{pure}\ (\texttt{lift}\ \rho) \ggg \mathcal{A}[\![C]\!] \ggg \texttt{pure}\ (\lambda s.\ \texttt{restrict}\ \rho\ s, p\ s)$$

Proof. By induction on the derivation of $\Gamma \vdash V : t$.

(FUN VAR) $\Gamma \vdash x : t$, so $x : t \in \Gamma$ and $\Sigma \vdash \rho(x) : t$. We have $\rho \vdash x \Rightarrow \mathbf{nil}, \rho(x)$, so $f = id$.

$$
\begin{aligned}
& \mathcal{A}[\![x]\!] \\
= \ & \texttt{pure}\ (\lambda s.\ (s, \mathcal{V}[\![x]\!]\ s)) \\
= \ & \texttt{pure}\ (\lambda s.\ (s, \texttt{lookup}\ x\ s)) \\
= \ & \texttt{pure}\ (\lambda s.\ (\texttt{restrict}\ \rho(\texttt{lift}\ \rho), p\ (\texttt{lift}\ \rho\ s))) \\
= \ & \texttt{lift}\ \rho \ggg (\lambda s.\ (\texttt{restrict}\ \rho\ s, p\ s)) \\
= \ & \texttt{lift}\ \rho \ggg \texttt{pure}\ id \ggg (\lambda s.\ (\texttt{restrict}\ \rho\ s, p\ s)) \\
= \ & \texttt{lift}\ \rho \ggg \mathcal{A}[\![x]\!] \ggg (\lambda s.\ (\texttt{restrict}\ \rho\ s, p\ s))
\end{aligned}
$$

(FUN CONST) $\Gamma \vdash c : ty(c)$. We have $\rho \vdash c \Rightarrow l \leftarrow c, l$, so $f = \lambda s.\ \texttt{add}\ l\ (s, c)$.

$$
\begin{aligned}
& \mathcal{A}[\![c]\!] \\
= \ & \texttt{pure}\ (\lambda s.\ s, c) \\
= \ & \texttt{pure}\ (\lambda s.\ \texttt{restrict}\ \rho(\texttt{lift}\ \rho\ s), l\ (\texttt{add}\ l\ (\texttt{lift}\ \rho\ s, c))) \\
= \ & \texttt{pure}\ (\texttt{lift}\ \rho) \ggg \texttt{pure}\ (\lambda s.\ \texttt{restrict}\ \rho\ s, l\ (\texttt{add}\ l\ (s, c))) \\
= \ & \texttt{pure}\ (\texttt{lift}\ \rho) \ggg \texttt{pure}\ (\lambda s.\ \texttt{add}\ l\ (s, c)) \ggg \texttt{pure}\ (\lambda s.\ \texttt{restrict}\ \rho\ s, l\ s) \\
= \ & \texttt{pure}\ (\texttt{lift}\ \rho) \ggg \mathcal{A}[\![l \leftarrow c]\!] \ggg \texttt{pure}\ (\lambda s.\ \texttt{restrict}\ \rho\ s, l\ s)
\end{aligned}
$$

(FUN PAIR) $\Gamma \vdash V_1, V_2 : t_1 * t_2$. We have $\rho \vdash V_1, V_2 \Rightarrow C_1; C_2, (p_1, p_2)$. By the IH, $\mathcal{A}[\![C_1]\!] = \texttt{pure}\ f_1$ and $\mathcal{A}[\![C_2]\!] = \texttt{pure}\ f_2$, where f_1 and f_2 are either id or add s. We also have:

$$
\begin{aligned}
& \mathcal{A}[\![Vi]\!] \\
= \ & \texttt{pure}\ (\lambda s.\ s, \mathcal{V}[\![Vi]\!]\ s) \\
= \ & \texttt{pure}\ (\texttt{lift}\ \rho) \ggg \mathcal{A}[\![C_i]\!] \ggg \texttt{pure}\ (\lambda s.\ \texttt{restrict}\ \rho\ s, p_i\ s) \\
= \ & \texttt{pure}\ (\texttt{lift}\ \rho) \ggg \texttt{pure}\ f_i \ggg \texttt{pure}\ (\lambda s.\ \texttt{restrict}\ \rho\ s, p_i\ s) \\
= \ & \texttt{pure}\ (\lambda s.\ \texttt{restrict}\ \rho(f_i(\texttt{lift}\ \rho\ s)), p_i\ (f_i\ (\texttt{lift}\ \rho\ s))) \\
= \ & \texttt{pure}\ (\lambda s.\ s, p_i\ (f_i\ (\texttt{lift}\ \rho\ s)))
\end{aligned}
$$

So $\mathcal{V}[\![Vi]\!]\ s = p_i(f_i(\texttt{lift}\ \rho\ s))$. Let $f = f_1; f_2$. We derive:

$$
\begin{aligned}
& \mathcal{A}[\![V_1, V_2]\!] \\
= \ & \texttt{pure}\ (\lambda s.\ s, (\mathcal{V}[\![V_1]\!]\ s, \mathcal{V}[\![V_2]\!]\ s)) \\
= \ & \texttt{pure}\ (\lambda s.\ s, (p_1\ (f_1\ (\texttt{lift}\ \rho\ s)), p_2\ (f_2\ (\texttt{lift}\ \rho\ s)))) \quad \text{by weakening/independence} \\
= \ & \texttt{pure}\ (\lambda s.\ s, (p_1\ ((f_1; f_2)(\texttt{lift}\ \rho\ s)), p_2\ ((f_1; f_2)(\texttt{lift}\ \rho\ s)))) \\
= \ & \texttt{pure}\ (\lambda s.\ \texttt{restrict}\ \rho\ (f_1; f_2(\texttt{lift}\ \rho\ s)), \\
& \qquad (p_1\ ((f_1; f_2)(\texttt{lift}\ \rho\ s)), p_2\ ((f_1; f_2)(\texttt{lift}\ \rho\ s)))) \\
= \ & \texttt{pure}\ (\texttt{lift}\ \rho) \ggg \texttt{pure}\ (f_1; f_2) \ggg \texttt{pure}\ (\lambda s.\ \texttt{restrict}\ \rho\ s, (p_1\ s, p_2\ s)) \\
= \ & \texttt{pure}\ (\texttt{lift}\ \rho) \ggg \mathcal{A}[\![C_1]\!] \ggg \mathcal{A}[\![C_2]\!] \ggg \texttt{pure}\ (\lambda s.\ \texttt{restrict}\ \rho\ s, (p_1, p_2)\ s) \\
= \ & \texttt{pure}\ (\texttt{lift}\ \rho) \ggg \mathcal{A}[\![C_1; C_2]\!] \ggg \texttt{pure}\ (\lambda s.\ \texttt{restrict}\ \rho\ s, (p_1, p_2)\ s)
\end{aligned}
$$

□

Restatement of Theorem 2 *$\Gamma \vdash M : t$ and $\Sigma \vdash \rho : \Gamma$ and $\rho \vdash M \Rightarrow C, p$ then:*

$$\mathcal{A}[\![M]\!] = \texttt{pure}\ (\texttt{lift}\ \rho) \ggg \mathcal{A}[\![C]\!] \ggg \texttt{pure}\ (\lambda s.\ (\texttt{restrict}\ \rho\ s, p\ s))$$

Proof. By induction on $\Gamma \vdash M : t$.

(FUN VAR) By the value lemma.

(FUN CONST) By the value lemma.

(FUN PAIR) By the value lemma.

(FUN OPERATOR) $\Gamma \vdash V_1 \otimes V_2 : b_3$ and $\rho \vdash V_1 \otimes V_2 \Rightarrow C_1; C_2, l_1 \otimes l_2$. We have $\mathcal{A}[\![V_1 \otimes V_2]\!] = \texttt{pure}\ (\lambda s.\ s, \mathcal{V}[\![V_1]\!]\ s \otimes \mathcal{V}[\![V_2]\!]\ s)$. By the value lemma (Lemma 16):

$$
\begin{aligned}
&\mathcal{A}[\![V_i]\!] \\
&= \texttt{pure}\ (\lambda s.\ s, \mathcal{V}[\![V_i]\!]\ s) \\
&= \texttt{pure}\ (\texttt{lift}\ \rho) \ggg \mathcal{A}[\![C_i]\!] \ggg \texttt{pure}\ (\lambda s.\ \texttt{restrict}\ \rho\ s, l_i\ s) \\
&= \texttt{pure}\ (\texttt{lift}\ \rho) \ggg \texttt{pure}\ f_i \ggg \texttt{pure}\ (\lambda s.\ \texttt{restrict}\ \rho\ s, l_i\ s) \\
&= \texttt{pure}\ (\lambda s.\ \texttt{restrict}\ \rho\ (f_i\ (\texttt{lift}\ \rho\ s)), l_i\ (f_i\ (\texttt{lift}\ \rho\ s))) \\
&= \texttt{pure}\ (\lambda s.\ s, l_i\ (f_i (\texttt{lift}\ \rho\ s))) \\
&= \texttt{pure}\ (\lambda s.\ s, l_i\ ((f_1; f_2)(\texttt{lift}\ \rho\ s)) \qquad \text{by weakening/independence}
\end{aligned}
$$

So $\mathcal{V}[\![V_i]\!]\ s = l_i\ ((f_1; f_2)(\texttt{lift}\ \rho\ s))$. We derive:

$$
\begin{aligned}
&\mathcal{A}[\![V_1 \otimes V_2]\!] \\
&= \texttt{pure}\ (\lambda s.\ s, \mathcal{V}[\![V_1]\!]\ s \otimes \mathcal{V}[\![V_2]\!]\ s) \\
&= \texttt{pure}\ (\lambda s.\ s, l_1\ ((f_1; f_2)(\texttt{lift}\ \rho\ s)) \otimes l_2\ ((f_1; f_2)(\texttt{lift}\ \rho\ s))) \\
&= \texttt{pure}\ (\texttt{lift}\ \rho) \ggg \texttt{pure}\ (f_1; f_2) \ggg \texttt{pure}\ (\lambda s.\ \texttt{restrict}\ \rho\ s, l_1\ s \otimes l_2\ s))) \\
&= \texttt{pure}\ (\texttt{lift}\ \rho) \ggg \mathcal{A}[\![C_1]\!] \ggg \mathcal{A}[\![C_2]\!] \ggg \texttt{pure}\ (\lambda s.\ \texttt{restrict}\ \rho\ s, l_1\ s \otimes l_2\ s))) \\
&= \texttt{pure}\ (\texttt{lift}\ \rho) \ggg \mathcal{A}[\![C_1; C_2]\!] \ggg \texttt{pure}\ (\lambda s.\ \texttt{restrict}\ \rho\ s, l_1\ s \otimes l_2\ s)))
\end{aligned}
$$

(FUN PROJ1) $\Gamma \vdash V.1 : t_1$ and $\Gamma \vdash V : t_1 * t_2$. We have $\rho \vdash V \Rightarrow C, (p_1, p_2)$ and $\rho \vdash V.1 \Rightarrow C, p_1$. By the value lemma as before, we can conclude $\mathcal{V}[\![V]\!]\ s = (p_1, p_2)\ (f\ (\texttt{lift}\ \rho\ s))$. Therefore:

$$
\begin{aligned}
&\mathcal{A}[\![V.1]\!] \\
&= \texttt{pure}\ (\lambda s.\ s, \texttt{fst}\ \mathcal{V}[\![V]\!]\ s) \\
&= \texttt{pure}\ (\lambda s.\ s, \texttt{fst}\ ((p_1, p_2)(f\ (\texttt{lift}\ \rho\ s))) \\
&= \texttt{pure}\ (\lambda s.\ s, p_1\ (f\ (\texttt{lift}\ \rho\ s)) \\
&= \texttt{pure}\ (\texttt{lift}\ \rho) \ggg \texttt{pure}\ f \ggg \texttt{pure}\ (\lambda s.\ \texttt{restrict}\ \rho\ s, p_1\ s) \\
&= \texttt{pure}\ (\texttt{lift}\ \rho) \ggg \mathcal{A}[\![C]\!] \ggg \texttt{pure}\ (\lambda s.\ \texttt{restrict}\ \rho\ s, p_1\ s)
\end{aligned}
$$

(FUN PROJ2) Symmetric to Proj1.

(FUN IF) $\Gamma \vdash \textbf{if}\ V_1\ \textbf{then}\ M_2\ \textbf{else}\ M_3 : t$. We have:

$$\rho \vdash \ldots \Rightarrow C_1; \textbf{if}\ l_1\ \textbf{then local}\ \text{locs}(C_2)\ \textbf{in}\ C_2; p \leftarrow 2\ \textbf{else local}\ \text{locs}(C_3)\ \textbf{in}\ C_3; p \leftarrow p_3, p$$

Our IHs are: $\mathcal{A}[\![M_i]\!] = \texttt{pure}\ (\texttt{lift}\ \rho) \ggg \mathcal{A}[\![C_i]\!] \ggg \texttt{pure}\ (\lambda s.\ \texttt{restrict}\ \rho\ s, p_i\ s)$. In the case of V_1, we actually have $\mathcal{A}[\![C_1]\!] = \texttt{pure}\ f_1 \mathcal{V}[\![V_1]\!]\ s = l_1(f_1(\texttt{lift}\ \rho\ s))$,

by the value lemma. We now calculate (at length):

$$
\begin{aligned}
&\mathcal{A}[\![\textbf{if } V_1 \textbf{ then } M_2 \textbf{ else } M_3]\!] \\
&= \texttt{choose}\ (\lambda s.\ \textbf{if } \mathcal{V}[\![V_1]\!]\ s \textbf{ then } \mathcal{A}[\![M_2]\!] \textbf{ else } \mathcal{A}[\![M_3]\!]) \\
&= \texttt{choose}\ (\lambda s.\ \textbf{if } l_1\ (f_1\ (\texttt{lift}\ \rho\ s)) \\
&\qquad \textbf{then } \texttt{pure}\ (\texttt{lift}\ \rho) \ggg \mathcal{A}[\![C_2]\!] \ggg \texttt{pure}\ (\lambda s.\ \texttt{restrict}\ \rho\ s, p_2\ s) \\
&\qquad \textbf{else } \texttt{pure}\ (\texttt{lift}\ \rho) \ggg \mathcal{A}[\![C_3]\!] \ggg \texttt{pure}\ (\lambda s.\ \texttt{restrict}\ \rho\ s, p_3\ s)) \\
&= \texttt{pure}\ (\texttt{lift}\ \rho) \ggg \texttt{choose}\ (\lambda s.\ \textbf{if } l_1\ (f_1\ s) \\
&\qquad \textbf{then } \mathcal{A}[\![C_2]\!] \ggg \texttt{pure}\ (\lambda s.\ \texttt{restrict}\ \rho\ s, p_2\ s) \\
&\qquad \textbf{else } \mathcal{A}[\![C_3]\!] \ggg \texttt{pure}\ (\lambda s.\ \texttt{restrict}\ \rho\ s, p_3\ s)) \\
&= \texttt{pure}\ (\texttt{lift}\ \rho) \ggg \texttt{choose}\ (\lambda s.\ \textbf{if } l_1\ (f_1\ s) \\
&\qquad \textbf{then } \mathcal{A}[\![C_2]\!] \ggg \mathcal{A}[\![p \leftarrow p_2]\!] \ggg \texttt{pure}\ (\lambda s.\ \texttt{restrict}\ \rho\ s, p\ s) \\
&\qquad \textbf{else } \mathcal{A}[\![C_3]\!] \ggg \mathcal{A}[\![p \leftarrow p_3]\!] \ggg \texttt{pure}\ (\lambda s.\ \texttt{restrict}\ \rho\ s, p\ s)) \\
&= \texttt{pure}\ (\texttt{lift}\ \rho) \ggg \texttt{choose}\ (\lambda s.\ \textbf{if } l_1\ (f_1\ s) \\
&\qquad \textbf{then } \mathcal{A}[\![C_2]\!] \ggg \mathcal{A}[\![p \leftarrow p_2]\!] \ggg \texttt{pure}\ (\texttt{drop}\ \mathrm{locs}(C_2)) \ggg \texttt{pure}\ (\lambda s.\ \texttt{restrict}\ \rho\ s, p\ s) \\
&\qquad \textbf{else } \mathcal{A}[\![C_3]\!] \ggg \mathcal{A}[\![p \leftarrow p_3]\!] \ggg \texttt{pure}\ (\texttt{drop}\ \mathrm{locs}(C_3)) \ggg \texttt{pure}\ (\lambda s.\ \texttt{restrict}\ \rho\ s, p\ s)) \\
&= \texttt{pure}\ (\texttt{lift}\ \rho) \ggg \texttt{choose}\ (\lambda s.\ \textbf{if } l_1\ (f_1\ s) \\
&\qquad \textbf{then } \mathcal{A}[\![C_2]\!] \ggg \mathcal{A}[\![p \leftarrow p_2]\!] \ggg \texttt{pure}\ (\texttt{drop}\ \mathrm{locs}(C_2)) \\
&\qquad \textbf{else } \mathcal{A}[\![C_3]\!] \ggg \mathcal{A}[\![p \leftarrow p_3]\!] \ggg \texttt{pure}\ (\texttt{drop}\ \mathrm{locs}(C_3))) \ggg \\
&\quad \texttt{pure}\ (\lambda s.\ \texttt{restrict}\ \rho\ s, p\ s) \\
&= \texttt{pure}\ (\texttt{lift}\ \rho) \ggg \mathcal{A}[\![C_1]\!] \ggg \texttt{choose}\ (\lambda s.\ \textbf{if } (l_1\ s) \\
&\qquad \textbf{then } \mathcal{A}[\![C_2; p \leftarrow p_2]\!] \ggg \texttt{pure}\ (\texttt{drop}\ \mathrm{locs}(C_2)) \\
&\qquad \textbf{else } \mathcal{A}[\![C_3; p \leftarrow p_3]\!] \ggg \texttt{pure}\ (\texttt{drop}\ \mathrm{locs}(C_3))) \ggg \\
&\quad \texttt{pure}\ (\lambda s.\ \texttt{restrict}\ \rho\ s, p\ s) \\
&= \texttt{pure}\ (\texttt{lift}\ \rho) \ggg \mathcal{A}[\![C_1]\!] \ggg \texttt{choose}\ (\lambda s.\ \textbf{if } (l_1\ s) \\
&\qquad \textbf{then } \mathcal{A}[\![\textbf{local } \mathrm{locs}(C_2) \textbf{ in } C_2; p \leftarrow p_2]\!] \\
&\qquad \textbf{else } \mathcal{A}[\![\textbf{local } \mathrm{locs}(C_3) \textbf{ in } C_3; p \leftarrow p_3]\!]) \\
&\quad \texttt{pure}\ (\lambda s.\ \texttt{restrict}\ \rho\ s, p\ s)
\end{aligned}
$$

(FUN LET) $\Gamma \vdash \textbf{let } x = M_1 \textbf{ in } M_2 : t_2$; by inversion, $\Gamma \vdash M_1 : t_1$ and $\Gamma, x : t_1 \vdash M_2 : t_2$.
Let $\rho' = \rho\{x \mapsto p_1\}$. We have:

$$
\begin{aligned}
&\rho \vdash M_1 \Rightarrow C_1, p_1 \\
&\rho' \vdash M_2 \Rightarrow C_2, p_2 \\
&\rho \vdash \textbf{let } x = M_1 \textbf{ in } M_2 \Rightarrow C_1; C_2, p_2
\end{aligned}
$$

As our IHs:

$$
\begin{aligned}
&\mathcal{A}[\![M_1]\!] = \texttt{pure}\ (\texttt{lift}\ \rho) \ggg \mathcal{A}[\![C_1]\!] \ggg \texttt{pure}\ (\lambda s.\ \texttt{restrict}\ \rho\ s, p_1 s) \\
&\mathcal{A}[\![M_2]\!] = \texttt{pure}\ (\texttt{lift}\ \rho') \ggg \mathcal{A}[\![C_2]\!] \ggg \texttt{pure}\ (\lambda s.\ \texttt{restrict}\ \rho' s, p_2 s)
\end{aligned}
$$

We derive:

$$
\begin{aligned}
&\mathcal{A}[\![\textbf{let } x = M_1 \textbf{ in } M_2]\!] \\
&= \mathcal{A}[\![M_1]\!] \ggg \mathtt{pure}\ (\mathtt{add}\ x) \ggg \mathcal{A}[\![M2]\!] \ggg \mathtt{pure}\ (\lambda s, y.\ \mathtt{drop}\ x\ s, y) \\
&= \mathtt{pure}\ (\mathtt{lift}\ \rho) \ggg \mathcal{A}[\![C_1]\!] \ggg \mathtt{pure}\ (\lambda s.\ \mathtt{restrict}\ \rho\ s, p_1\ s) \ggg \\
&\quad \mathcal{A}[\![M2]\!] \ggg \mathtt{pure}\ (\lambda s, y.\ \mathtt{drop}\ x\ s, y) \\
&= \mathtt{pure}\ (\mathtt{lift}\ \rho) \ggg \mathcal{A}[\![C_1]\!] \ggg \mathtt{pure}\ (\lambda s.\ \mathtt{restrict}\ \rho\ s, p_1\ s) \ggg \\
&\quad \mathtt{pure}\ (\mathtt{add}\ x) \ggg \mathtt{pure}\ (\mathtt{lift}\ \rho') \ggg \mathcal{A}[\![C_2]\!] \ggg \mathtt{pure}\ (\lambda s.\ \mathtt{restrict}\ \rho' s, p_2\ s) \ggg \\
&\quad \mathtt{pure}\ (\lambda s, y.\ \mathtt{drop}\ x\ s, y) \\
&= \mathtt{pure}\ (\mathtt{lift}\ \rho) \ggg \mathcal{A}[\![C_1]\!] \ggg \mathcal{A}[\![C_2]\!] \ggg \mathtt{pure}\ (\lambda s.\ \mathtt{restrict}\ \rho' s, p_2\ s) \ggg \\
&\quad \mathtt{pure}\ (\lambda s, y.\ \mathtt{drop}\ x\ s, y) \\
&= \mathtt{pure}\ (\mathtt{lift}\ \rho) \ggg \mathcal{A}[\![C_1]\!] \ggg \mathcal{A}[\![C_2]\!] \ggg \mathtt{pure}\ (\lambda s.\ \mathtt{restrict}\ \rho\ s, p_2\ s) \\
&= \mathtt{pure}\ (\mathtt{lift}\ \rho) \ggg \mathcal{A}[\![C_1; C_2]\!] \ggg \mathtt{pure}\ (\lambda s.\ \mathtt{restrict}\ \rho\ s, p_2\ s)
\end{aligned}
$$

(FUN RANDOM) $\Gamma \vdash \textbf{random}(D(V)) : b$, where $D : (b_1, ..., b_n) -> b_{n+1}, \Gamma \vdash V : (b_1, ..., b_n)$.
We have $\rho \vdash V \Rightarrow C, p$ and $\rho \vdash D(V) \Rightarrow C; l \leftarrow D(p), l$. By the value lemma, $\mathcal{A}[\![C]\!] = \mathtt{pure}\ f$ and $\mathcal{V}[\![V]\!]\ s = p\ (f\ (\mathtt{lift}\ \rho\ s))$. We derive:

$$
\begin{aligned}
&\mathcal{A}[\![\textbf{random}(D(V))]\!] \\
&= \mathtt{extend}\ (\lambda s.\ \mu_{D(\mathcal{V}[\![V]\!]\ s)}) \\
&= \mathtt{extend}\ (\lambda s.\ \mu_{D(p(f(\mathtt{lift}\ \rho\ s)))}) \\
&= \mathtt{pure}\ (\mathtt{lift}\ \rho) \ggg \mathtt{extend}\ (\lambda s.\ \mu_{D(p(fs))}) \ggg \mathtt{pure}\ (\lambda s, v.\ \mathtt{restrict}\ \rho\ s, v) \\
&= \mathtt{pure}\ (\mathtt{lift}\ \rho) \ggg \mathtt{pure}\ f \ggg \mathtt{extend}\ (\lambda s.\ \mu_{D(p\ s)}) \ggg \mathtt{pure}\ (\lambda s, v.\ \mathtt{restrict}\ \rho\ s, v) \\
&= \mathtt{pure}\ (\mathtt{lift}\ \rho) \ggg \mathcal{A}[\![C]\!] \ggg \mathtt{extend}\ (\lambda s.\ \mu_{D(p\ s)}) \ggg \mathtt{pure}\ (\lambda s, v.\ \mathtt{restrict}\ \rho\ s, v) \\
&= \mathtt{pure}\ (\mathtt{lift}\ \rho) \ggg \mathcal{A}[\![C]\!] \ggg \mathtt{extend}\ (\lambda s.\ \mu_{D(p\ s)}) \ggg \\
&\quad \mathtt{pure}\ (\mathtt{add}\ l) \ggg \mathtt{pure}\ (\lambda s.\ \mathtt{restrict}\ \rho\ s, l\ s) \\
&= \mathtt{pure}\ (\mathtt{lift}\ \rho) \ggg \mathcal{A}[\![C; l \leftarrow D(p)]\!] \ggg \mathtt{pure}\ (\lambda s.\ \mathtt{restrict}\ \rho\ s, l\ s)
\end{aligned}
$$

(FUN OBSERVE) $\Gamma \vdash \textbf{observe}\ V : \textbf{unit}$ and $\Gamma \vdash V : b$ for some base type b. We have $\rho \vdash V \Rightarrow C, l$. By the value lemma: $\mathcal{A}[\![C]\!] = \mathtt{pure}\ f$ and $\mathcal{V}[\![V]\!]\ s = l\ (f\ (\mathtt{lift}\ \rho\ s))$.

$$
\begin{aligned}
&\mathcal{A}[\![\textbf{observe}\ V]\!] \\
&= \mathtt{observe}\ (\lambda s.\ \mathcal{V}[\![V]\!]\ s) \ggg \mathtt{pure}\ (\lambda s.\ (s, ())) \\
&= \mathtt{observe}\ (\lambda s.\ l(f(\mathtt{lift}\ \rho\ s))) \ggg \mathtt{pure}\ (\lambda s.\ s, ()) \\
&= \mathtt{pure}\ (\mathtt{lift}\ \rho) \ggg \mathtt{observe}\ (\lambda s.\ l\ (f\ s)) \ggg \mathtt{pure}\ (\lambda s.\ \mathtt{restrict}\ \rho\ s, ()\ s) \\
&= \mathtt{pure}\ (\mathtt{lift}\ \rho) \ggg \mathtt{pure}\ f \ggg \mathtt{observe}\ (\lambda s.\ l\ s) \ggg \mathtt{pure}\ (\lambda s.\ \mathtt{restrict}\ \rho\ s, ()\ s) \\
&= \mathtt{pure}\ (\mathtt{lift}\ \rho) \ggg \mathcal{A}[\![C]\!] \ggg \mathtt{observe}\ (\lambda s.\ l\ s) \ggg \mathtt{pure}\ (\lambda s.\ \mathtt{restrict}\ \rho\ s, ()\ s) \\
&= \mathtt{pure}\ (\mathtt{lift}\ \rho) \ggg \mathcal{A}[\![C; \mathtt{observe}\ l]\!] \ggg \mathtt{pure}\ (\lambda s.\ \mathtt{restrict}\ \rho\ s, ()\ s)
\end{aligned}
$$

□

A.3 Proof of Theorem 3

Restatement of Theorem 3 *If $\Sigma \vdash C : \Sigma'$ and $\mu \in \mathtt{M}\langle \mathtt{S}\langle\Sigma\rangle\rangle$ is compatible with $\mathcal{I}[\![C]\!]$ then $\mathcal{I}[\![C]\!]\ \mu\ A = \mathcal{J}[\![\mathcal{G}[\![C]\!]]\!]^{\Sigma'}_{\Sigma}\ \mu\ A$ for all measurable $A \in \mathcal{M}_{\mathtt{S}\langle\Sigma,\Sigma'\rangle}$.*

Proof. By induction on $\Sigma \vdash C : \Sigma'$.
We write λ_Γ for the standard (Lebesgue or counting) measure over $\mathtt{S}\langle\Gamma\rangle$.
We write $s(l) \triangleq \mathtt{lookup}\ l\ \ s$. By abuse of notation, we let $\nu \times \lambda_\varepsilon = \nu$ and $\lambda = \lambda_{\Sigma'}$.

Base cases:

(1) $\Sigma \vdash \mathbf{nil} : \varepsilon$. We have

$$\mathcal{I}[\![\mathbf{nil}]\!]\ \mu\ A = (\mathtt{pure}\ \mathtt{id})\ \mu\ A = \mu(\mathtt{id}^{-1}(A)) = \mu(A)$$

and

$$\mathcal{J}[\![\mathcal{G}[\![\mathbf{nil}]\!]]\!]^{\Sigma'}_{\Sigma}\ \mu\ A = \int_A \mathcal{J}[\![\varnothing]\!]\, d\mu = \int_A d\mu = \mu(A)$$

(2) $\Sigma \vdash l \leftarrow c : \Sigma'$. Here $\Sigma' = l{:}\mathrm{ty}(c)$ and $A \in \mathcal{M}_{\Sigma, l:\mathrm{ty}(c)}$
Let $B := \{\mathtt{drop}\ \{l\}\ s \mid s \in A \wedge s(l) = c\}$.

$$\begin{aligned}
& & \mathcal{I}[\![l \leftarrow c]\!]\ \mu\ A \\
\text{(by def)} & = & (\mathtt{pure}\ \lambda s.\mathtt{add}\ l\ (s,c))\ \mu\ A \\
\text{(by def of pure)} & = & \mu((\lambda s.\mathtt{add}\ l\ (s,c))^{-1}(A)) \\
\text{(by simpl)} & = & \mu(B)
\end{aligned}$$

and

$$\begin{aligned}
& & \mathcal{J}[\![\mathcal{G}[\![l \leftarrow c]\!]]\!]^{\Sigma'}_{\Sigma}\ \mu\ A \\
\text{(by def)} & = & \int_A \delta(s(l) = c)\, d(\mu \times \lambda)(s) \\
\text{(by Tonelli's theorem)} & = & \int \left(\int_{\{s \mid \mathtt{add}\ l\ (s,x) \in A\}} \delta(x = c)\, d\mu \right) d\lambda(x) \\
\text{(by simpl)} & = & \int \delta(x = c) \left(\int_{\{s \mid \mathtt{add}\ l\ (s,x) \in A\}} d\mu \right) d\lambda(x) \\
\text{(by def of } \delta) & = & \int_{\{s \mid \mathtt{add}\ l\ (s,c) \in A\}} d\mu \\
\text{(by def of } B) & = & \int_B d\mu \\
\text{(by def)} & = & \mu(B)
\end{aligned}$$

(3) $\Sigma \vdash l \leftarrow l' : \Sigma'$. As (2).
(4) $\Sigma \vdash l \leftarrow l_1 \otimes l_2 : \Sigma'$. As (2).
(5) $\Sigma \vdash l \xleftarrow{s} D(l_1, \ldots, l_n) : \Sigma'$. Here $\Sigma' = l{:}\mathrm{ty}(D)$,

$$\begin{aligned}
& & \mathcal{I}[\![l \xleftarrow{s} D(l_1, \ldots, l_n)]\!]\ \mu\ A \\
\text{(by def)} & = & (\mathtt{extend}\ \lambda s.\mu_{D(s(l_1),\ldots,s(l_n))} \ggg \mathtt{add}\ l)\ \mu\ A \\
\text{(by def of } \ggg, \mathtt{add}\ l) & = & \mathtt{extend}\ \lambda s.\mu_{D(s(l_1),\ldots,s(l_n))}\ \mu\ \{(s', y) \mid \mathtt{add}\ l\ (s', y) \in A\} \\
\text{(by def of extend)} & = & \int_{\mathtt{drop}\ l\ A} \mu_{D(s'(l_1),\ldots,s'(l_n))}(\{y \mid (\mathtt{add}\ l\ (s', y) \in A\})\, d\mu(s')
\end{aligned}$$

and

$$\begin{aligned}
& \mathcal{J}[\![\mathcal{G}[\![l \leftarrow c]\!]]\!]_{\Sigma}^{\Sigma'}\ \mu\ A \\
(\text{by def}) \quad &= \int_A (D(s(l_1),\ldots,s(l_n))\ x)\, d(\mu \times \lambda)(s,x) \\
(\text{by Tonelli's theorem}) \quad &= \int_{\mathtt{drop}\ l\ A} \left(\int_{\{x \mid \mathtt{add}\ l\ (s',x) \in A\}} D(s(l_1),\ldots,s(l_n))\ x\, d\lambda(x) \right) d\mu(s) \\
(\text{by def of } \mu_D) \quad &= \int_{\mathtt{drop}\ l\ A} \mu_{D(s(l_1),\ldots,s(l_n))}(\{x \mid \mathtt{add}\ l\ (s,x) \in A\})\, d\mu(s)
\end{aligned}$$

(6) $\Sigma \vdash \textbf{observe}\ l : \Sigma'$. We let $B = \{s \in \mathtt{S}\langle\Sigma\rangle \mid s(l) = 0\}$. There are two cases.

(a) : $\Sigma \vdash l : \textbf{real}$. By compatibility there is a function m that is a density of μ on some neighbourhood B' of B. Then

$$\begin{aligned}
& \mathcal{I}[\![\textbf{observe}\ l]\!]\ \mu\ A \\
(\text{by def}) \quad &= (\mathtt{observe}\ \mathtt{lookup}\ l\ )\ \mu\ A \\
(\text{by def}) \quad &= \dot{\mu}[A || \mathtt{lookup}\ l = 0.0] \\
(\text{by additivity}) \quad &= \dot{\mu}[A \cap B' || \mathtt{lookup}\ l = 0.0] + \dot{\mu}[A \setminus B' || \mathtt{lookup}\ l = 0.0] \\
(\text{by simpl}) \quad &= \dot{\mu}[A \cap B' || \mathtt{lookup}\ l = 0.0] \\
(\text{by compatibility}) \quad &= \int_{A \cap B'} m(s) \cdot \delta(s(l)) d\lambda(s)
\end{aligned}$$

and

$$\begin{aligned}
& \mathcal{J}[\![\mathcal{G}[\![\textbf{observe}\ l]\!]]\!]_{\Sigma}^{\Sigma'}\ \mu\ A \\
(\text{by def}) \quad &= \mathcal{J}[\![\mathsf{Constant}_{0.0}(l)]\!]_{\Sigma}^{\Sigma'}\ \mu\ A \\
(\text{by def}) \quad &= \int_A \delta(s(l) = 0.0)\, d\mu(s) \\
(\text{by additivity}) \quad &= \int_{A \cap B'} \delta(s(l)) d\mu(s) + \int_{A \setminus B'} \delta(s(l)) d\mu(s) \\
(\text{by simpl}) \quad &= \int_{A \cap B'} \delta(s(l)) d\mu(s) \\
(\text{by density}) \quad &= \int_{A \cap B'} m(s) \cdot \delta(s(l)) d\lambda(s)
\end{aligned}$$

(b) Otherwise, $\Sigma \vdash l : b$ for some $b \neq \textbf{real}$ and

$$\mathcal{I}[\![\textbf{observe}\ l]\!]\ \mu\ A = (\mathtt{observe}\ \mathtt{lookup}\ l\ )\ \mu\ A = \mu(A \cap B)$$

and

$$\begin{aligned}
& \mathcal{J}[\![\mathcal{G}[\![\textbf{observe}\ l]\!]]\!]_{\Sigma}^{\Sigma'}\ \mu\ A \\
(\text{by def}) \quad &= \mathcal{J}[\![\mathsf{Constant}_{0_b}(l)]\!]_{\Sigma}^{\Sigma'}\ \mu\ A \\
(\text{by def}) \quad &= \int_A [s(l) = 0_b]\, d\mu(s) \\
(\text{by def of Iverson brackets}) \quad &= \int_{A \cap B} 1\, d\mu + \int_{A \setminus B} 0\, d\mu \\
(\text{by simpl}) \quad &= \mu(A \cap B)
\end{aligned}$$

Induction cases:

(1) $\Sigma \vdash \textbf{local}\ l : b\ \textbf{in}\ C : \Sigma'$. Here $\Sigma \vdash C : \Sigma''$ and $\Sigma' = \Sigma'' \setminus \{l : b\}$. We consider the case where $\Sigma', l : b = \Sigma''$; other cases can be treated by permuting the dimensions of the space $\mathtt{S}\langle \Sigma', l : b\rangle$.
Then

$$\begin{aligned} & \mathcal{I}[\![I]\!]\ \mu\ A \\ (\text{by def of } \mathtt{choose}) \quad &= \quad (\mathcal{I}[\![C]\!] \ggg \mathtt{pure}\ \mathtt{drop}\ \{l\})\ \mu\ A \\ (\text{by def of } \ggg, \mathtt{pure}, \mathtt{drop}\ \cdot) \quad &= \quad \mathcal{I}[\![C]\!]\ \mu\ (A \times V_b)\,. \end{aligned}$$

By induction, for all measurable A' and compatible measures μ' we have $\mathcal{I}[\![C]\!]\ \mu'\ A' = \mathcal{J}[\![\mathcal{G}[\![C]\!]]\!]^{\Sigma''}_{\Sigma}\ \mu'\ A'$, so

$$\begin{aligned} & \mathcal{J}[\![\mathsf{new}\ l : b\ \mathsf{in}\ \mathcal{G}[\![C]\!]]\!]^{\Sigma'}_{\Sigma}\ \mu\ A \\ (\text{by def}) \quad &= \quad \int_A \mathcal{J}[\![\mathsf{new}\ l : b\ \mathsf{in}\ \mathcal{G}[\![C]\!]]\!]\, d(\mu \times \lambda_{\Sigma'}) \\ (\text{by def}) \quad &= \quad \int_A \left(\int_{V_b} \mathcal{J}[\![\mathcal{G}[\![C]\!]]\!]\, d\lambda_b \right) d(\mu \times \lambda_{\Sigma'}) \\ (\text{by Tonelli's theorem}) \quad &= \quad \int_{A \times V_b} \mathcal{J}[\![\mathcal{G}[\![C]\!]]\!]\, d(\mu \times \lambda_{\Sigma'} \times \lambda_b) \\ (\text{by simpl}) \quad &= \quad \int_{A \times V_b} \mathcal{J}[\![\mathcal{G}[\![C]\!]]\!]\, d(\mu \times \lambda_{\Sigma''}) \\ (\text{by induction}) \quad &= \quad \mathcal{I}[\![C]\!]\ \mu\ (A \times V_b) \end{aligned}$$

(2) $\Sigma \vdash \textbf{if}\ l\ \textbf{then}\ C_1\ \textbf{else}\ C_2 : \Sigma'$. Here $\Sigma \vdash C_i : \Sigma'$. By induction, for all measurable A_1, A_2 and compatible measures μ_1, μ_2 we have $\mathcal{I}[\![C_1]\!]\ \mu_1\ A_1 = \mathcal{J}[\![\mathcal{G}[\![C_1]\!]]\!]^{\Sigma'}_{\Sigma}\ \mu_1\ A_1$ and $\mathcal{I}[\![C_2]\!]\ \mu_2\ A_2 = \mathcal{J}[\![\mathcal{G}[\![C_2]\!]]\!]^{\Sigma'}_{\Sigma}\ \mu_2\ A_2$.
Let $B = \{s \in \mathtt{S}\langle\Sigma\rangle \mid s(l) = \textbf{true}\}$ and $B' = \{s \in \mathtt{S}\langle\Sigma, \Sigma'\rangle \mid s(l) = \textbf{true}\}$. Let $\nu_1 = \mu|_B$ and $\nu_2 = \mu|_{\overline{B}}$. Then $\mathcal{I}[\![I]\!]\ \mu\ A) = (\mathcal{I}[\![C_1]\!]\ \nu_1\ A) + (\mathcal{I}[\![C_2]\!]\ \nu_2\ A)$ by the definition of `choose`, and

$$\mathcal{J}[\![\mathsf{Gate}(l, \mathcal{G}[\![C_1]\!], \mathcal{G}[\![C_2]\!])]\!]^{\Sigma'}_{\Sigma}\ \mu\ A$$

(by def)

$$= \int_A \prod_{i=1,2} (\mathcal{J}[\![\mathcal{G}[\![C_i]\!]]\!]\ s)^{[\neg^i \neg s(l)]}\, d(\mu \times \lambda_{\Sigma'})(s)$$

(by additivity and $\mu = \nu_1 + \nu_2$)

$$= \int_A \prod_{i=1,2} (\mathcal{J}[\![\mathcal{G}[\![C_i]\!]]\!]\ s)^{[\neg^i \neg s(l)]}\, d(\nu_1 \times \lambda_{\Sigma'})(s) + \int_A \prod_{i=1,2} (\mathcal{J}[\![\mathcal{G}[\![C_i]\!]]\!]\ s)^{[\neg^i \neg s(l)]}\, d(\nu_2 \times \lambda_{\Sigma'})(s)$$

(by simplification)

$$= \int_A \mathcal{J}[\![\mathcal{G}[\![C_1]\!]]\!]\, d(\nu_1 \times \lambda_{\Sigma'}) + \int_A \mathcal{J}[\![\mathcal{G}[\![C_2]\!]]\!]\, d(\nu_2 \times \lambda_{\Sigma'})$$

(by induction)

$$= \mathcal{I}[\![C_1]\!]\ \nu_1\ (A \times V_{\Sigma'}) + \mathcal{I}[\![C_2]\!]\ \nu_2\ (A \times V_{\Sigma'})$$

(3) Assume that $\Sigma \vdash C_1;C_2 : \Sigma'$ by $\Sigma \vdash C_1 : \Sigma_1$ and $\Sigma;\Sigma_1 \vdash C_2 : \Sigma'$ with $\Sigma' = \Sigma_1,\Sigma_2$. By induction, for all measurable A_1,A_2 and compatible measures μ_1,μ_2 we have $\mathcal{I}[\![C_1]\!]\ \mu_1\ A_1 = \mathcal{J}[\![\mathcal{G}[\![C_1]\!]]\!]^{\Sigma_1}_{\Sigma}\ \mu_1\ A_1$ and $\mathcal{I}[\![C_2]\!]\ \mu_2\ A_2 = \mathcal{J}[\![\mathcal{G}[\![C_2]\!]]\!]^{\Sigma_2}_{\Sigma;\Sigma_1}\ \mu_2\ A_2$.

We first show that

$$\int f\, d(\mathcal{I}[\![C_1]\!]\ \mu) = \int f \cdot \mathcal{J}[\![\mathcal{G}[\![C_1]\!]]\!]\ d(\mu \times \lambda_{\Sigma_1}) \tag{2}$$

for all non-negative Lebesgue-integrable ($d(\mathcal{I}[\![C_1]\!]\ \mu)$) functions f. Let $f_1, f_2, \ldots$ be a series of non-negative simple functions converging to f from below. We assume that $f_i = \sum_{0<j\leq n_i} a_{ij} \cdot I_{ij}$ where the I_{ij} are indicator functions for the measurable sets S_{ij}. We then get

$$\begin{aligned}
\int f\, d(\mathcal{I}[\![C_1]\!]\ \mu) &= \lim_{i\to\infty} \int f_i\, d(\mathcal{I}[\![C_1]\!]\ \mu) \\
\text{(by def of } f_i) \quad &= \lim_{i\to\infty} \int \sum_{0<j\leq n_i} a_{ij} \cdot I_{ij}\, d(\mathcal{I}[\![C_1]\!]\ \mu) \\
\text{(by simpl)} \quad &= \lim_{i\to\infty} \sum_{0<j\leq n_i} a_{ij} \cdot (\mathcal{I}[\![C_1]\!]\ \mu\ S_{ij}) \\
\text{(by induction)} \quad &= \lim_{i\to\infty} \sum_{0<j\leq n_i} a_{ij} \cdot (\mathcal{J}[\![\mathcal{G}[\![C_1]\!]]\!]^{\Sigma_1}_{\Sigma}\ \mu\ S_{ij}) \\
\text{(by def)} \quad &= \lim_{i\to\infty} \sum_{0<j\leq n_i} a_{ij} \cdot \int_{S_{ij}} \mathcal{J}[\![\mathcal{G}[\![C_1]\!]]\!]\ d(\mu \times \lambda_{\Sigma_1}) \\
\text{(by simpl)} \quad &= \lim_{i\to\infty} \int \sum_{0<j\leq n_i} a_{ij} \cdot I_{ij} \cdot \mathcal{J}[\![\mathcal{G}[\![C_1]\!]]\!]\ d(\mu \times \lambda_{\Sigma_1}) \\
\text{(by def of } f_i) \quad &= \lim_{i\to\infty} \int f_i \cdot \mathcal{J}[\![\mathcal{G}[\![C_1]\!]]\!]\ d(\mu \times \lambda_{\Sigma_1}) \\
\text{(by the monotone convergence theorem)} \quad &= \int f \cdot \mathcal{J}[\![\mathcal{G}[\![C_1]\!]]\!]\ d(\mu \times \lambda_{\Sigma_1})
\end{aligned}$$

Using eq. (2), we now derive

$$
\begin{aligned}
& \mathcal{J}[\![\mathcal{G}[\![C_1;C_2]\!]]\!]_{\Sigma}^{\Sigma'}\ \mu\ A \\
(\text{by def}) \quad &= \int_A \mathcal{J}[\![\mathcal{G}[\![C_1;C_2]\!]]\!]\ d(\mu \times \lambda_{\Sigma'}) \\
(\text{by def}) \quad &= \int_A \mathcal{J}[\![\mathcal{G}[\![C_1]\!]]\!]\ \mathcal{J}[\![\mathcal{G}[\![C_2]\!]]\!]\ d(\mu \times \lambda_{\Sigma_1} \times \lambda_{\Sigma_2}) \\
(\text{by Tonelli's theorem}) \quad &= \int_{A|_{\Sigma;\Sigma_1}} (\mathcal{J}[\![\mathcal{G}[\![C_1]\!]]\!]\ s) \left(\int_{\{s' \in \mathbb{S}\langle\Sigma_2\rangle \mid s \cup s' \in A\}} \mathcal{J}[\![\mathcal{G}[\![C_2]\!]]\!]\ d\lambda_{\Sigma_2} \right) d(\mu \times \lambda_{\Sigma_1})(s) \\
(\text{by eq. (2)}) \quad &= \int_{A|_{\Sigma;\Sigma_1}} \left(\int_{\{s' \in \mathbb{S}\langle\Sigma_2\rangle \mid s \cup s' \in A\}} \mathcal{J}[\![\mathcal{G}[\![C_2]\!]]\!]\ d\lambda_{\Sigma_2} \right) d(\mathcal{I}[\![C_1]\!]\ \mu)(s) \\
(\text{by Tonelli's theorem}) \quad &= \int_A \mathcal{J}[\![\mathcal{G}[\![C_2]\!]]\!]\ d((\mathcal{I}[\![C_1]\!]\ \mu) \times \lambda_{\Sigma_2}) \\
(\text{by induction}) \quad &= \int_A d(\mathcal{I}[\![C_2]\!]\ (\mathcal{I}[\![C_1]\!]\ \mu) \\
(\text{by simpl}) \quad &= \mathcal{I}[\![C_2]\!]\ (\mathcal{I}[\![C_1]\!]\ \mu)\ A \\
(\text{by def of } \ggg) \quad &= (\mathcal{I}[\![C_1]\!] \ggg \mathcal{I}[\![C_2]\!])\ \mu\ A
\end{aligned}
$$

B Implementation of measure transformers, finite case

We implement finite measures over type α with finite support (α Dist) as finite maps, when the type α has comparison operators. The function dAdd computes the union of two measures. The functions dFold, dFlatMap and dFilter are standard. As a base case, pointMass w a creates a point mass of measure w at a.

Measures as finite maps

```
type (α) Dist when α:comparison = (α,float) Map

type (α,β) DTrans when α:comparison and β:comparison = (α Dist) →(β Dist)

let dFold (f:α →float →β) (g: β→β→β) (s : β) (d:α Dist)=
    Map.fold (fun acc k v →g acc (f k v)) s d

let dAdd (acc:β Dist) (m:β Dist) =
  let addOne (s: β Dist) k v =
    match Map.tryFind k s with
    | Some(v') →Map.add k (v+v') s
    | None →Map.add k v s in
  Map.fold addOne acc m

let dFlatMap (f:α →float →β Dist) = dFold f dAdd Map.empty

let dFilter (p:α →bool) = (Map.filter (fun a _ →p a)):(α,α) DTrans

let pointMass w (a:α) = (Map.add a w Map.empty):α Dist
```

We define the delta, bernoulli and uniform distributions in terms of pointMasses and union. Function pairDists computes the independent product of two distributions. We lift a function of type $\alpha \to \beta$ Dist (cf. Kleisli arrows of the finite measure monad) to a distribution transformer by rescaleing its value at each element of the input distribution by the measure of that element. The function getWeight computes $|\mu|$.

Helper functions

```
let delta a = pointMass 1.0 a
let discreteUniform (k:int):int Dist = List.reduce dAdd [for x in [ 1 .. k ] → pointMass
    (1.0/(float k)) x]
let bernoulli (p:float) = dAdd (pointMass p true) (pointMass (1.0−p) false)

let pairDists (da:α Dist) (db:β Dist) =
  dFlatMap (fun a wa → dFlatMap (fun b wb → pointMass (wa∗wb) (a,b)) db) da

let rescale (r:float) = dFlatMap (fun a w → pointMass (w∗r) a)
let lift (f:α→β Dist) = dFlatMap (fun a w → rescale w (f a))

let getWeight (da:α Dist) = dFold (fun _ w → w) (+) 0.0 da
```

The implementation of the measure transformer combinators is in most cases by lifting an appropriate function from α to α Dist. Observation is directly implemented using filtering.

Measure Transformer Combinators

```
let (>>>): (α,β) DTrans →(β,γ) DTrans →(α,γ) DTrans =
    fun t u d →u(t(d))
let pure: (α→β) →(α,β) DTrans =
    fun f →lift (fun a →delta (f a))
let (***): (α,β) DTrans →(α,γ) DTrans →(α,β∗γ) DTrans =
    fun t u d →pairDists (t(d)) (u(d))
let extend:(α →β Dist)→ (α,α∗β) DTrans =
    fun f →lift (fun a →pairDists (delta a) (f a))
let weight: float →(α,α) DTrans =
    fun r →lift (fun a →pointMass r a)
let choose: (α →(α,β) DTrans) →(α,β) DTrans =
    fun f →lift (fun a →f a (delta a))
let observe: (α→bool) →(α,α) DTrans = dFilter

(∗ bind of "Unnormalized Probability" Monad; return = delta ∗)
let (>>=) m f = (extend f >>> pure snd) m
```

We can express the Monty Hall problem directly via (some of) these combinators. The function montyHall is parameterized by a pick function, which is called when the guess of the player is correct, and the host thus has a choice of two doors to open. The function pick should then return a distribution describing the action of the host. We then evaluate the program for three different values of pick, and can see if the player is ever at a disadvantage when switching doors.

Monty Hall problem

```
let montyHall pick = (* Door of the car *)
  extend (fun _ → discreteUniform 3)
  >>> (* Door of the player *)
  extend (fun _ → discreteUniform 3)
  >>> (* Host picks a door different from c,p *)
  extend (fun ((_,c),p) → if c = p then pick c else delta (6−(c+p)))
  >>> (* Should player swap to remaining door? *)
  pure (fun (((_,c),p),h) → c <> p && c <> h)

let unif cp = (discreteUniform 2) >>= (fun f → delta (1 + (cp+f−1)%3))
let cycl cp = delta (1 + cp%3)
let least cp = delta (if cp = 1 then 2 else 1)

let t1 = Map.tryFind true (montyHall unif (delta ()))
let f1 = Map.tryFind false (montyHall unif (delta ()))

let t2 = Map.tryFind true (montyHall cycl (delta ()))
let f2 = Map.tryFind false (montyHall cycl (delta ()))

let t3 = Map.tryFind true (montyHall least (delta ()))
let f3 = Map.tryFind false (montyHall least (delta ()))
```

In this implementation, we use a value language also including records and tagged unions. A state is an association list from locations to values. Locations are untyped. An implementation in for instance Coq or Agda could use dependent types to directly enforce well-typedness of states and programs at the cost of some complexity (cf. [3]). The functions get and drop apply to the most recently added occurrence of the location.

States and operations

```
type Location = string
type Value = | IntVal of int | FloatVal of float | BoolVal of bool
             | UnitVal | LeftVal of Value | RightVal of Value
             | PairVal of Value * Value | RecVal of (string*Value) list
type State = (Location*Value) list
let add l ((s:State),v) = ((l,v)::s):State
let get l s = snd (List.find (fun (l',_) → l=l') s)
let drop l (s:State) =
  match s with
  | (l',v)::s' → if l=l' then s' else (l',v)::(drop l s')
  | [] → s
```

We show here the different clauses of the measure tranformer semantics of an extended version of Fun. Note that this is not the compiler backend itself. An expression

yields a measure transformer from state measures to measures over the return value and final state. The measure transformer corresponding to a function additionally uses the argument to the function. Finally, a distribution taking argument of type α is denoted by a function from α to Value Dist. The semantics of Fun values and expressions correspond to those given earlier in the paper, except that we here do not require A-normal form.

Semantics of Fun plus sum types and procedures

```
type vDenot = (State → Value)
type eDenot = (State,State∗Value) DTrans
type fDenot = (State∗Value,State∗Value) DTrans
type α mDenot = (α → Value Dist) (∗ Type of α −parametrized distribution ∗)

(∗ Semantics of values ∗)
let var (l:Location) (s:State) = get l s
let constant (c:Value) (s:State) = c
let leftVal (v:vDenot) (s:State) = LeftVal (v s)
let rightVal (v:vDenot) (s:State) = RightVal (v s)
let pairVal (v1:vDenot) v2 (s:State) = PairVal ((v1 s),(v2 s))

(∗ Semantics of expressions ∗)
let valExp (v:vDenot) = pure (fun s → s,v s)
let firstExp (e:eDenot) = e >>> (pure (fun (e',(PairVal (f,s))) → e',f))
let secondExp (e:eDenot) = e >>> (pure (fun (e',(PairVal (f,s))) → e',s))
let letExp (x:Location) (e1:eDenot) (e2:eDenot) =
  e1 >>> (pure (add x)) >>> e2 >>> (pure (first (drop x)))
let obsExp (e:eDenot) =
  e >>> (observe (fun (_,BoolVal b) → b)) >>> (pure (second (fun _ → UnitVal)))
let caseExp (e:eDenot) (l:fDenot) (r:fDenot) =
    e >>> (choose (fun(_,v) →
        match v with
        | LeftVal _ → (pure (second (fun (LeftVal v) → v))) >>> l
        | RightVal _ → (pure (second (fun (RightVal v) → v))) >>> r
        ))
let randomExp (m:Value mDenot) = extend (fun (_,x) → m x) >>> (pure (first fst))
let appExp (f:fDenot) (e:eDenot) = e >>> f
let funExp (x:Location) (e:eDenot) = (pure (add x)) >>> e >>> (pure (first (drop x)))
```

The measure transformer semantics of Imp follow the definitions in Section 4.1.

Semantics of Imp

```
type cDenot = (State,State) DTrans (* Type of denotation of a compound stmt *)
let nil:cDenot = pure id
let loc (l:Location) (s:State) = get l s
let const (c:Value) (s:State) = c
let pthen (i:cDenot) (c:cDenot) = i >>> c
let assign (l:Location) (e:vDenot) = pure (fun s → add l (s,e s))
let sampl (l:Location) (m:State mDenot) = (extend m) >>> (pure (add l))
let ifthenelse (e:vDenot) (c1:cDenot) (c2:cDenot) =
  choose (fun s → if (e s = BoolVal(true)) then c1 else c2)
let obs (l:Location) = observe (fun s → get l s = BoolVal(true))
```

References

1. M. Abadi and P. Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *J. Cryptology*, 15(2):103–127, 2002.
2. P. Audebaud and C. Paulin-Mohring. Proofs of randomized algorithms in Coq. *Sci. Comput. Program.*, 74(8):568–589, 2009.
3. G. Barthe, B. Grégoire, and S. Z. Béguelin. Formal certification of code-based cryptographic proofs. In *POPL*, pages 90–101. ACM, 2009.
4. P. Billingsley. *Probability and Measure*. Wiley, 3rd edition, 1995.
5. K. A. Bonawitz. *Composable Probabilistic Inference with Blaise*. PhD thesis, MIT, 2008. Available as Technical Report MIT-CSAIL-TR-2008-044.
6. J. Borgström, A. D. Gordon, M. Greenberg, J. Margetson, and J. Van Gael. Measure transformer semantics for Bayesian machine learning. Technical Report MSR–TR–2011–18, Microsoft Research, 2011.
7. N. N. Dalvi, C. Ré, and D. Suciu. Probabilistic databases: diamonds in the dirt. *Commun. ACM*, 52(7):86–94, 2009.
8. M. Erwig and S. Kollmansberger. Functional pearls: Probabilistic functional programming in Haskell. *J. Funct. Program.*, 16(1):21–34, 2006.
9. D. A. S. Fraser, P. McDunnough, A. Naderi, and A. Plante. On the definition of probability densities and sufficiency of the likelihood map. *J. Probability and Mathematical Statistics*, 15:301–310, 1995.
10. N. Goodman, V. K. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum. Church: a language for generative models. In *UAI*, pages 220–229. AUAI Press, 2008.
11. T. Graepel, J. Q. Candela, T. Borchert, and R. Herbrich. Web-scale Bayesian click-through rate prediction for sponsored search advertising in Microsoft's Bing search engine. In *International Conference on Machine Learning*, pages 13–20, 2010.
12. V. Gupta, R. Jagadeesan, and P. Panangaden. Stochastic processes as concurrent constraint programs. In *POPL*, pages 189–202, 1999.
13. R. Herbrich, T. Minka, and T. Graepel. TrueSkill(TM): A Bayesian skill rating system. In *Advances in Neural Information Processing Systems 20*, 2007.
14. J. Hurd. *Formal verification of probabilistic algorithms*. PhD thesis, University of Cambridge, 2001. Available as University of Cambridge Computer Laboratory Technical Report UCAM–CL–TR–566, May 2003.
15. E. T. Jaynes. *Probability Theory: The Logic of Science*, chapter 15.7 The Borel-Kolmogorov paradox, pages 467–470. CUP, 2003.
16. C. Jones and G. D. Plotkin. A probabilistic powerdomain of evaluations. In *LICS*, pages 186–195. IEEE Computer Society, 1989.
17. O. Kiselyov and C. Shan. Embedded probabilistic programming. In *Domain-Specific Languages*, pages 360–384, 2009.
18. O. Kiselyov and C. Shan. Monolingual probabilistic programming using generalized coroutines. In *UAI*, 2009.
19. D. Koller and N. Friedman. *Probabilistic Graphical Models*. The MIT Press, 2009.
20. D. Koller, D. A. McAllester, and A. Pfeffer. Effective Bayesian inference for stochastic programs. In *AAAI/IAAI*, pages 740–747, 1997.
21. D. Kozen. Semantics of probabilistic programs. *J. Comput. Syst. Sci.*, 22(3):328–350, 1981.
22. F. R. Kschischang, B. J. Frey, and H.-A. Loeliger. Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory*, 47(2):498–519, 2001.
23. M. Z. Kwiatkowska, G. Norman, and D. Parker. Quantitative analysis with the probabilistic model checker PRISM. *ENTCS*, 153(2):5–31, 2006.

24. G. Lowe. Quantifying information flow. In *CSFW*, pages 18–31. IEEE Computer Society, 2002.
25. D. J. C. MacKay. *Information Theory, Inference, and Learning Algorithms*. CUP, 2003.
26. A. McCallum, K. Schultz, and S. Singh. FACTORIE: Probabilistic programming via imperatively defined factor graphs, 2009. Poster at 23rd Annual Conference on Neural Information Processing Systems (NIPS).
27. A. McIver and C. Morgan. *Abstraction, refinement and proof for probabilistic systems*. Monographs in computer science. Springer, 2005.
28. F. McSherry. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *SIGMOD Conference*, pages 19–30. ACM, 2009.
29. T. Mhamdi, O. Hasan, and S. Tahar. On the formalization of the Lebesgue integration theory in HOL. In *Interactive Theorem Proving (ITP 2010)*, 2010.
30. T. Minka, J. Winn, J. Guiver, and A. Kannan. Infer.NET 2.3, Nov. 2009. Software available from http://research.microsoft.com/infernet.
31. T. Minka and J. M. Winn. Gates. In *NIPS*, pages 1073–1080. MIT Press, 2008.
32. T. P. Minka. Expectation Propagation for approximate Bayesian inference. In *UAI*, pages 362–369. Morgan Kaufmann, 2001.
33. I. Ntzoufras. *Bayesian Modeling Using WinBUGS*. Wiley, 2009.
34. P. Panangaden. *Labelled Markov processes*. Imperial College Press, 2009.
35. S. Park, F. Pfenning, and S. Thrun. A probabilistic language based upon sampling functions. In *POPL*, pages 171–182. ACM, 2005.
36. A. Pfeffer. IBAL: A probabilistic rational programming language. In B. Nebel, editor, *IJCAI*, pages 733–740. Morgan Kaufmann, 2001.
37. A. Pfeffer. *Statistical Relational Learning*, chapter The design and implementation of IBAL: A General-Purpose Probabilistic Language. MIT Press, 2007.
38. A. Radul. Report on the probabilistic language scheme. In *Proceedings of the 2007 symposium on Dynamic languages*, pages 2–10. ACM, 2007.
39. N. Ramsey and A. Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *POPL*, pages 154–165, 2002.
40. J. Reed and B. C. Pierce. Distance makes the types grow stronger: A calculus for differential privacy. In *ICFP*, pages 157–168, 2010.
41. J. S. Rosenthal. *A First Look at Rigorous Probability Theory*. World Scientific, 2nd edition, 2006.
42. N. Saheb-Djahromi. Probabilistic LCF. In *MFCS*, volume 64 of *LNCS*, pages 442–451. Springer, 1978.
43. D. Syme, A. Granicz, and A. Cisternino. *Expert F#*. Apress, 2007.
44. J. Winn and T. Minka. Probabilistic programming with Infer.NET. Machine Learning Summer School lecture notes, available at http://research.microsoft.com/~minka/papers/mlss2009/, 2009.
45. J. M. Winn and C. M. Bishop. Variational message passing. *Journal of Machine Learning Research*, 6:661–694, 2005.
46. E. S. Yudkowsky. An intuitive explanation of Bayesian reasoning, 2003. Available at http://yudkowsky.net/rational/bayes.