FAKULTÄT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Computer Science

A Verified Compiler for Probability Density Functions

Manuel Eberl

FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Computer Science

A Verified Compiler for Probability Density Functions

Verifikation eines Compilers für Wahrscheinlichkeitsdichten

Author:	Manuel Eberl
Supervisor:	Prof Tobias Nipkow, PhD
Advisors:	Dr Johannes Hölzl
Date:	October 15, 2014

I assure the single-handed composition of this Master's Thesis, only supported by declared resources.

Garching, 10 October 2014 Manuel Eberl

Abstract

Bhat et al. [BBGR13] developed an inductive compiler that computes density functions for probability spaces described by programs in a probabilistic functional language. In this thesis, we implement such a compiler for a modified version of this language within the theorem prover Isabelle and give a formal proof of its soundness w. r. t. the semantics of the source and target language.

Zusammenfassung

Bhat et al. [BBGR13] entwickelten einen induktiven Compiler zur automatischen Berechnung von Dichten bestimmter Wahrscheinlichkeitsräume. Diese werden durch Programme in einer probabilistischen funktionalen Sprache beschrieben. In dieser Arbeit wird solch ein Compiler für eine ähnliche Sprache in dem Theorembeweiser Isabelle implementiert und seine Korrektheit bezüglich der Semantik der zugrundeliegenden Ausgangs- und Zielsprache formal bewiesen.

Contents

Acknowledgements

1 Introduction

1.1 Motivation

Random distributions of practical significance can often be expressed as probabilistic functional programs. A simple example would be a board game in which a die is cast and, depending on the result, more dice are cast and their result influences the game in some way.

For instance, consider the Pen & Paper Role Playing Game *Call of Cthulhu*: players have a number of *skills* such as *Archæology*, *Library Usage*, etc. A player's proficiency in a skill is a number between 0 and 100, and when a player uses a skill in an adventure, she must roll a d100 (a die with numbers from 1 to 100). If the result is below her proficiency in that skill, she succeeds; if not, she fails. A result between 1 and 5 or below one fifth of her proficiency is a *critical success*, resulting in her receiving a chance to increase her proficiency in the skill: she rolls another d100 and if the result is *above* her proficiency, the skill proficiency is permanently increased by an amount determined through a d10 roll. This can be expressed concisely as a probabilistic program in the following way:

$$
\begin{array}{l}
\textbf{let}\ x\ =\ \text{UniformInt}\ (1, 100) \\
\textbf{in}\ \ \textbf{if}\ x > \text{skill}\ \textbf{then} \\
\qquad (\text{FAILURE}, \text{skill}) \\
\quad\ \ \textbf{else if}\ x \le 5\ \vee\ x \cdot 5 \le \text{skill}\ \textbf{then} \\
\qquad \textbf{let}\ \text{increase}\ =\ \textbf{if}\ \text{UniformInt}\ (1, 100) > \text{skill}\ \textbf{then}\ \text{UniformInt}\ (1, 10)\ \textbf{else}\ 0 \\
\qquad \textbf{in}\ \ (\text{CRITICAL\_SUCCESS}, \text{skill} + \text{increase}) \\
\quad\ \ \textbf{else} \\
\qquad (\text{SUCCESS}, \text{skill})
\end{array}
$$

Finite and discrete examples such as this can be studied with brute force, i. e. simply summing the probabilities for every single case, but more complex random experiments involving an infinite number of outcomes (e. g. Poisson distribution) or continuous distributions require a more refined approach.

When studying a random distribution such as this, it is often desirable to determine its *probability density function* (PDF). This can be used to e. g. determine the expectation or sample the distribution with a sampling method such as MCMC.

In 2013, Bhat et al. presented a compiler that computes the probability distribution function of a program in the probabilistic functional language *Fun*. [BBGR13] They evaluated the compiler on a number of practical problems and concluded that it reduces the amount of time and effort required to model them in an MCMC system significantly compared to hand-written models.

Bhat et al. also stated that their eventual goal is the formal verification of such a compiler in a theorem prover [BAVG12]. This has the advantage of providing *guaranteed correctness*, i. e. the result of the compilation is provably a PDF for the source expression, according to the formal semantics. This greatly increases the confidence one can have in the compiler.

In this Master's thesis, we implemented such a compiler for a similar probabilistic functional language in the interactive theorem prover *Isabelle/HOL* and formally proved its correctness.

1.2 Related work

As mentioned before, this work is based on the work of Bhat et al. [BAVG12, BBGR13], in which a density compiler for probability spaces described by expressions in the language *Fun*. This is a small functional language with basic arithmetic, Boolean logic, product and sum types, conditionals, and a number of builtin discrete and continuous distributions. It does, however, not support lists or recursion. The correctness proof is purely pen and paper, but the authors stated that formalisation in a proof assistant such as Coq is the ultimate goal. [BAVG12]

pGCL, a probabilistic language, *has* been formalised with a proof assistant – first in HOL by Hurd et al. [HMM05], then in Isabelle/HOL by David Cock [Coc12, Coc14]. pGCL contains a rich set of language features, such as recursion and probabilistic and nondeterministic choice. David Cock formally proved a large number of results about the semantics of the language and developed mechanisms for refinement and program verification; however, the focus of this work was verification of probabilistic programs, not compiling them to density functions. Bhat et al. mention that reconciling recursion with probability density functions is difficult [BAVG12], so a feature-rich language such as pGCL is probably not suited for developing a density compiler.

Audebaud and Paulin-Mohring [APM06] also implemented a probabilistic functional language with recursion in a theorem prover (namely Coq). Their focus was also on verification of probabilistic programs. While Cock uses a shallow embedding in which any type and operation of the surrounding logic of the theorem prover (i. e. HOL) can be used, Audebaud and Paulin-Mohring use a deep embedding with a restricted type system, expression structure, predefined primitive functions, etc. Like Bhat et al. and we, they also explicitly reference the *Giry monad* as a basis for their work.

1.3 Utilised tools

As stated before, we use the interactive theorem prover Isabelle/HOL. Isabelle is a generic proof assistant that can be used with a number of logical frameworks (object logics) such as *Higher Order Logic* (HOL) or *Zermelo-Fraenkel set theory* (ZF). The most widely used instance is Isabelle/HOL, which is what we use.

We heavily rely on Johannes Hölzl's Isabelle formalisation of measure theory [Hö12], which is already part of the Isabelle/HOL library. We also use Sudeep Kanav's formalisation of the Gaussian distribution and a number of libraries from the proof of the Central Limit Theorem by Avigad et al. [AHS14], namely the notion of interval and set integrals and the Fundamental Theorem of Calculus for these integrals. All of these have been or will be moved to the Isabelle/HOL library by the maintainers as well.

1.4 Outline

In section 2.1, we will explain the notation we will use and then give a brief overview of the mathematical basics we require – in particular the Giry Monad – in section 2.2.

Section 3 contains the definition and the semantics of the source and target language. Section 4 defines the abstract compiler and gives a high-level outline of the soundness proof. Section 5 then explains the refinement of the abstract compiler to the concrete compiler and the final correctness result and evaluates the compiler on a simple example.

Section 6 gives an overview of how much work went into the different parts of the project and what difficulties we encountered. It also lists possible improvements and other future work before summarising the result we obtained. Finally, the appendix contains a table of all notation and auxiliary functions used in this work for reference.

2 Preliminaries

2.1 Notation

In the following two sections, we will give more detailed explanations of the most important notation we will use. For a full overview of all the nonstandard notation used, see the appendix.

2.1.1 Deviations from standard mathematical notation

In order to maintain coherence with the notation used in Isabelle, we will deviate from standard mathematical notation in the following ways:

- As is convention in functional programming (and therefore in Isabelle), function application is often written as $f\ x$ instead of $f(x)$. It is the operation that binds strongest and it associates to the left, i. e. $f\ x + 1$ is $(f\ x) + 1$ and $f\ g\ x$ is $(f\ g)\ x$.
- The integral over integrand f with variable x and the measure μ is written as

$$\int x.\ f\ x\ \partial\mu \quad \text{instead of} \quad \int f(x)\,\mathrm{d}\mu(x)$$

- The special notion of a *nonnegative integral*[1] is used; this integral "ignores" the negative part of the integrand. Effectively:

$$\int^{+} x.\ f\ x\ \partial\mu \quad \hat{=} \quad \int \max(0, f(x))\,\mathrm{d}\mu(x)$$

- Lambda abstractions, such as $\lambda x.\ x^2$, are used to specify functions. The corresponding standard mathematical notation would be $x \mapsto x^2$.
- Some additional, less important differences in notation are explained in the appendix, section A.1.

2.1.2 Semantics notation

We will always use Γ to denote a type environment, i. e. a function from variable names to types, and σ to denote a state, i. e. a function from variable names to values. Note that variable names are always natural numbers as we use de Bruijn indices. The letter Υ (Upsilon) will be used to denote *density contexts*, which are used in the compiler.

We also employ the notation $t \bullet \Gamma$ resp. $v \bullet \sigma$ to denote the insertion of a new variable with the type t (resp. value v) into a typing environment (resp. state). This is used when entering the scope of a bound variable; the newly inserted variable then has index 0 and all other variables are incremented by 1. The precise definition is as follows:[2]

[1] This is a very central concept in the measure theory library in Isabelle. We will mostly use it with nonnegative functions anyway, so the distinction is purely formal.

[2] Note the analogy to the notation $x \# xs$ for prepending an element to a list. This is because contexts with de Bruijn indices are generally represented by lists, where the n-th element is the entry for the variable n, and inserting a value for a newly bound variable is then simply the prepending operation

$$(y \bullet f)(x) = \begin{cases} y & \text{if } x = 0 \\ f(x-1) & \text{otherwise} \end{cases}$$

We use the same notation for inserting a new variable into a set of variables, shifting all other variables, i. e.:

$$x \bullet V = \{x\} \cup \{y + 1 \mid y \in V\}$$

In general, the notation $\Gamma \vdash e \ : \ t$ and variations thereof will always mean "The expression e has type t in the type environment Γ", whereas $\Upsilon \vdash e : f$ and variations thereof mean "The expression e compiles to f under the context Υ". For an overview of these predicates, see the appendix, section A.2.

2.2 Mathematical basics

The category theory part of this section is based mainly on a presentation by Ernst-Erich Doberkat [Dob08]. For a more detailed introduction, see his textbook [Dob07] or the original paper by Michèle Giry [Gir82].

2.2.1 Subprobability spaces

A subprobability space is a measurable space $(A, \mathcal{A})$ with a measure μ such that every set $X \in \mathcal{A}$ has a measure ≤ 1, or, equivalently, $\mu(A) \leq 1$.

For technical reasons, we also assume $A \neq \emptyset$. This is required later in order to define the bind operation in the Giry monad in a convenient way within Isabelle. This non-emptiness condition will always be trivially satisfied by all the measure spaces used in this work.

2.2.2 The category Meas

Note that:

- For any measurable space $(A, \mathcal{A})$, the identity function is $\mathcal{A}$-$\mathcal{A}$-measurable.
- For any measurable spaces $(A, \mathcal{A})$, $(B, \mathcal{B})$, $(C, \mathcal{C})$, an $\mathcal{A}$-$\mathcal{B}$-measurable function f, and a $\mathcal{B}$-$\mathcal{C}$-measurable function g, the function $g \circ f$ is $\mathcal{A}$-$\mathcal{C}$-measurable.

Therefore, measurable spaces form a category Meas where:

- the objects of the category are measurable spaces
- the morphisms of the category are measurable functions
- the identity morphism $\mathbf{1}_{(A,\mathcal{A})}$ is the identity function $\mathrm{id}_A : A \to A,\ x \mapsto x$
- morphism composition is function composition

2.2.3 Kernel space

The kernel space $\mathbb{S}(A, \mathcal{A})$ of a measurable space $(A, \mathcal{A})$ is the natural measurable space over the measures over $(A, \mathcal{A})$ with a certain property. For our purposes, this property will be that they are subprobability measures (as defined in section 2.2.1).

Additionally, a natural property the kernel space should satisfy is that measuring a fixed set $X \in \mathcal{A}$ while varying the measure within $\mathbb{S}(A, \mathcal{A})$ should be a Borel-measurable function; formally:

$$\text{For all } X \in \mathcal{A},\ f \colon \mathbb{S}(A, \mathcal{A}) \to \mathbb{R},\ (B, \mathcal{B}, \mu) \mapsto \mu(X) \ \text{ is } \mathbb{S}(A, \mathcal{A})\text{-Borel-measurable}$$

We can now simply define $\mathbb{S}(A, \mathcal{A})$ as the smallest measurable space with the carrier set

$$M := \{\mu \mid \mu \text{ is a measure on } (A, \mathcal{A}),\ \mu(A) \leq 1\}$$

that fulfils this property, i. e. we let $\mathbb{S}(A, \mathcal{A}) := (M, \mathcal{M})$ with

$$\mathcal{M} := \{(\mu \mapsto \mu(X))^{-1}(Y) \mid X \in \mathcal{A}, Y \in \mathcal{B}\}$$

where $\mathcal{B}$ is the Borel algebra on $\mathbb{R}$.

Additionally, for a measurable function f, we define $\mathbb{S}(f) = \mu \mapsto f(\mu)$, where $f(\mu)$ denotes the pushforward measure (or image measure)[3]. Then $\mathbb{S}$ maps objects of Meas to objects of Meas and morphisms of Meas to morphisms of Meas. We can thus see that $\mathbb{S}$ is an endofunctor in the category Meas, as $(\mathrm{id}_{(A,\mathcal{A})})(\mu) = \mu$ and $(f \circ g)(\mu) = f(g(\mu))$.

2.2.4 Giry monad

The Giry monad naturally captures the notion of choosing a value according to a (sub-)probability distribution, using it as a parameter for another distribution, and observing the result.

Consequently, return (or η) yields a Dirac measure, i. e. a probability measure in which all the "probability" lies in a single element, and bind (or $\gg\!=$) integrates over all the input values to compute one single output measure. Formally, for measurable spaces $(A, \mathcal{A})$ and $(B, \mathcal{B})$, a measure μ on $(A, \mathcal{A})$, a value $x \in A$, and a $\mathcal{A}$-$\mathbb{S}(B, \mathcal{B})$-measurable function f:

$$\mathrm{return}_{(A,\mathcal{A})}\, x := X \mapsto \begin{cases} 1 & \text{if } x \in X \\ 0 & \text{otherwise} \end{cases} \qquad\qquad \mu \gg\!= f := X \mapsto \int x.\, f(x)(X)\, \partial\mu$$

Unfortunately, restrictions due to Isabelle's type system require us to determine the σ algebra of the resulting measurable space for bind $M\ f$, since this information cannot be provided by the type of f. This can be done by an additional parameter, but it is more convenient to define bind in such a way that it chooses an arbitrary value $x \in M$ and takes the σ algebra of $f\ x$ (or the count space on $\emptyset$ if M is empty)[4].

[3] $f(\mu) = X \mapsto \mu(f^{-1}(X))$.

[4] Note that for any $\mathcal{A} - \mathbb{S}(B, \mathcal{B})$-measurable function f, the σ algebra thus obtained is independent of which value is chosen, by definition of the kernel space.

This choice is somewhat nonstandard, but the difference is of no practical significance as we will not use bind on empty measure spaces.

To simplify the proofs for bind, we instead define the join operation (also known as μ in category theory) first and use it to then define bind. The join operation "flattens" objects, i. e. it maps an element of $\mathbb{S}(\mathbb{S}(A, \mathcal{A}))$ to one of $\mathbb{S}(A, \mathcal{A})$. Such an operation can be naturally defined as:

$$\text{join } \mu = X \mapsto \int \mu'(X)\, \partial\mu$$

Note that in Isabelle, join has an additional explicit parameter for the measurable space of the result to avoid the problem we had in bind. This makes expressions containing join more complicated; this is, however, justified by the easier proofs and will be unproblematic later since we will never use join directly, only bind.

bind can be defined using join in the following way, modulo handling of empty measure spaces[5]:

$$\mu \ggg f = \text{join } (\mathbb{S}(f)(\mu)) = \text{join } (f(\mu))$$

The coherence conditions

$$\text{join} \circ \mathbb{S}(\text{join}) = \text{join} \circ \text{join}$$

$$\text{join} \circ \mathbb{S}(\text{return}_{\mathbb{S}(A,\mathcal{A})}) = \text{join} \circ \text{return}_{\mathbb{S}(\mathbb{S}(A,\mathcal{A}))} = \mathbf{1}_{\mathbb{S}(A,\mathcal{A})}$$

$$\text{join} \circ \mathbb{S}(\mathbb{S}(f)) = \mathbb{S}(f) \circ \text{join}$$

can be proven easily by unfolding the definitions of the operations and applying the rules for integrals on image measures.

2.2.5 The monadic "do" syntax

For better readability, we employ a Haskell-style **do** notation for operations in the Giry monad. The notation is defined as follows:

$$\textbf{do } \{M\} \;\hat{=}\; M \qquad\qquad \textbf{do } \{x \leftarrow M;\ \langle\text{pattern}\rangle\} \;\hat{=}\; M \ggg (\lambda x.\ \textbf{do } \{\langle\text{pattern}\rangle\})$$

Example:

$$\textbf{do } \{\, x \leftarrow M;\ y \leftarrow f\ x;\ g\ x\ y\} \quad\hat{=}\quad M \ggg (\lambda x.\ f\ x \ggg (\lambda y.\ g\ x\ y))$$

[5] $\mathbb{S}(f)$, i. e. lifting a function on values to a function on measure spaces, is done by the distr function in Isabelle. This function was already defined in the library, which simplifies our proofs about bind, seeing as some of the proof work has already been done in the proofs about distr.

3 Language Syntax and Semantics

The source language used in the formalisation was modelled after the language *Fun* described by Bhat et al. [BBGR13]; similarly, the target language is almost identical to the target language used by Bhat et al. However, we have made the following changes in our languages:

- Variables are represented by de Bruijn indices to avoid handling freshness, capture-avoiding substitution, and related problems.
- No sum types are supported. Consequently, the **match** ... **with** ... command is replaced with an **IF** ... **THEN** ... **ELSE** ... command. Furthermore, booleans are a primitive type rather than represented as **unit** + **unit**.
- The type **double** is called **real** and it represents a real number with absolute precision as opposed to an IEEE 754 floating point number.
- Beta and Gamma distributions are not included.
- Gaussian distributions are parametrised with the standard deviation σ instead of the precision.

In the following sections, we give the precise syntax, typing rules, and semantics of both our source language and our target language.

3.1 Types, values, and operators

The source language and the target language share the same type system and the same operators. Figure 2 shows the types, values, and operators that exist in our languages. Figure 1 defines the semantics of the operators. Note that *bool*, *nat*, *int*, and *real* stand for the Isabelle/HOL types for booleans, natural numbers, integers, and reals, respectively.

All operators are *total*, meaning that for every input value of their parameter type, they return a single value of their result type. This requires some nonstandard definitions for nontotal operations such as division, logarithms, and square roots. Nontotality could also be handled by implementing operators in the Giry monad as well and letting them return either a Dirac distribution with a single result or, when evaluated for a parameter on which they are not defined, the null measure; however, this would significantly complicate many proofs.

To increase readability, we will use the abbreviations shown in the operator semantics table (Figure 1) and the following additional ones:

- $t_1 \times t_2$ stands for PRODUCT t_1 t_2
- TRUE and FALSE stand for BoolVal True and BoolVal False, respectively.
- RealVal, IntVal, etc. will be omitted in expressions when their presence is implicitly clear from the context
- $a - b$ stands for $a + (-b)$ and a/b for $a \cdot b^{-1}$

Operator	Notation	Input type	Output type	Semantics
Add	$a+b$	INTEG $\times$ INTEG	INTEG	$a+b$
		REAL $\times$ REAL	REAL	
Minus	$-a$	INTEG	INTEG	$-a$
		REAL	REAL	
Mult	$a \cdot b$	INTEG $\times$ INTEG	INTEG	$a \cdot b$
		REAL $\times$ REAL	REAL	
Inverse	a^{-1}	REAL	REAL	$\begin{cases} \frac{1}{a} & \text{for } a \neq 0 \\ 0 & \text{otherwise} \end{cases}$
Sqrt	$\sqrt{a}$	REAL	REAL	$\begin{cases} \sqrt{a} & \text{for } a \geq 0 \\ 0 & \text{otherwise} \end{cases}$
Exp	$\exp a$	REAL	REAL	e^a
Ln	$\ln a$	REAL	REAL	$\begin{cases} \ln a & \text{for } a > 0 \\ 0 & \text{otherwise} \end{cases}$
Pi	π	UNIT	REAL	π
Pow	$a \wedge b$	INTEG $\times$ INTEG	INTEG	$\begin{cases} a^b & \text{for } a \neq 0, b \geq 0 \\ 0 & \text{otherwise} \end{cases}$
		REAL $\times$ INTEG	REAL	
Fact	$a!$	INTEG	INTEG	$a!$
And	$a \wedge b$	BOOL $\times$ BOOL	BOOL	$a \wedge b$
Or	$a \vee b$	BOOL $\times$ BOOL	BOOL	$a \vee b$
Not	$\neg a$	BOOL	BOOL	$\neg a$
Equals	$a = b$	$t \times t$	BOOL	$a = b$
Less	$a < b$	INTEG $\times$ INTEG	BOOL	$a < b$
		REAL $\times$ REAL	BOOL	
Fst	fst z	$t_1 \times t_2$	t_1	a for $z = (a, b)$
Snd	snd z	$t_1 \times t_2$	t_2	b for $z = (a, b)$
Cast REAL	$\langle a \rangle$	BOOL	REAL	$\begin{cases} 1 & \text{for } a = \text{TRUE} \\ 0 & \text{otherwise} \end{cases}$
	implicit	INTEG	REAL	a as a real number
Cast INTEG		BOOL	INTEG	$\begin{cases} 1 & \text{for } a = \text{TRUE} \\ 0 & \text{otherwise} \end{cases}$
		REAL	INTEG	$\lfloor a \rfloor$

Figure 1: The types, abbreviations and semantics for the operators

datatype pdf_type $=$
UNIT | BOOL | INTEG | REAL | PRODUCT *pdf_type pdf_type*
datatype val $=$
UnitVal | BoolVal *bool* | IntVal *int* | RealVal *real* | $<\!|val, val|\!>$
datatype pdf_operator $=$
Fst | Snd | Add | Mult | Minus | Less | Equals | And | Or | Not | Pow |
Fact | Sqrt | Exp | Ln | Inverse | Pi | Cast *pdf_type*

Figure 2: The types and values in the source and target language

3.2 Auxiliary definitions

A number of auxiliary definitions are used in the definition of the semantics; for a full list of auxiliary functions see the appendix. The following two notions require a detailed explanation:

3.2.1 Measure embeddings

A *measure embedding* is the measure space obtained by "tagging" values in a measure space M with some injective function f (in fact, f will always be a datatype constructor). For instance, a set of values of type REAL can naturally be measured by stripping away the RealVal constructor and using a measure on real numbers (e. g. the Lebesgue-Borel measure) on the resulting set of reals. Formally:

$$\text{embed\_measure}\ M\ f\ =\ \Big(f(M),\ \{f(X) \mid X \in \mathcal{A}\},\ \lambda X.\ \mu(f^{-1}(X) \cap A)\Big)$$

3.2.2 Stock measures

The *stock measure* for a type t is the "natural" measure on values of that type. This is defined as follows:

- For the countable types UNIT, BOOL, and INTEG: the count measure over the corresponding type universes
- For REAL: the embedding of the Lebesgue-Borel measure on $\mathbb{R}$ with RealVal
- For $t_1 \times t_2$: the embedding of the product measure
  $$\text{stock\_measure}\ t_1 \otimes \text{stock\_measure}\ t_2$$
  with $\lambda(v, w).\ <\!|v, w|\!>$

Note that in order to save space and increase readability, we will often write ∂t instead of ∂ stock_measure t in integrals.

Using the stock measure, we can also construct a measure on *states* in the context of a typing environment Γ. A state on the variables V is a function that maps a variable in V to a value. A state σ is *well-formed* w. r. t. to V and Γ if it maps every variable $x \in V$ to a value v of type $\Gamma\ x$ and every variable $\notin V$ to undefined.

We now fix Γ and a finite V and consider the set of well-formed states w. r. t. V and Γ. Another representation of these states are tuples in which the i-th component is the value of the i-th variable in V. The natural measure that can be given to such tuples is then the finite product measure of the stock measures of the types of the variables, i. e.

$$\text{state\_measure}\ \Gamma\ V := \bigotimes_{x \in V} \text{stock\_measure}\ (\Gamma\ x)$$

3.3 Source language

Figures 3 and 4 show the syntax resp. the typing rules of the source language. Figure 6 defines the source language semantics as a primitively recursive function. Similarly to the abbreviations mentioned in section 3.1, we will omit Val when its presence is implicitly obvious from the context; e. g. if in some context, e is an expression and c is a constant real number, we will write $e + \text{Val}\ (\text{RealVal}\ c)$ as $e + c$.

Figure 5 shows the builtin distributions of the source language, their parameter type and domain, the type of the random variable they describe, and their density functions in terms of their parameter. When given a parameter *outside* their domain, they return the null measure.

$$\begin{aligned}&\textbf{datatype}\ \text{expr}\ =\\ &\qquad \text{Var}\ \textit{nat} \mid \text{Val}\ \textit{val} \mid \text{LET}\ \textit{expr}\ \text{IN}\ \textit{expr} \mid \textit{pdf\_operator}\ \$\ \textit{expr} \mid <\textit{expr}, \textit{expr}> \mid\\ &\qquad \text{Random}\ \textit{pdf\_dist} \mid \text{IF}\ \textit{expr}\ \text{THEN}\ \textit{expr}\ \text{ELSE}\ \textit{expr} \mid \text{Fail}\ \textit{pdf\_type}\end{aligned}$$

Figure 3: The source language syntax

$$\frac{}{\Gamma \vdash \text{Val } v \,:\, \text{val\_type } v}\;\text{ET\_VAL} \qquad \frac{}{\Gamma \vdash \text{Var } x \,:\, \Gamma\, x}\;\text{ET\_VAR} \qquad \frac{}{\Gamma \vdash \text{Fail } t \,:\, t}\;\text{ET\_FAIL}$$

$$\frac{\Gamma \vdash e \,:\, t \qquad \text{op\_type } op\ t = \text{Some } t'}{\Gamma \vdash op \,\$\, e \,:\, t'}\;\text{ET\_OP} \qquad \frac{\Gamma \vdash e_1 \,:\, t_1 \qquad \Gamma \vdash e_2 \,:\, t_2}{\Gamma \vdash <e_1, e_2> \,:\, t_1 \times t_2}\;\text{ET\_PAIR}$$

$$\frac{\Gamma \vdash e \,:\, \text{dist\_param\_type } dst}{\Gamma \vdash \text{Random } dst\ e \,:\, \text{dist\_result\_type } dst}\;\text{ET\_RAND} \qquad \frac{\Gamma \vdash b \,:\, \text{BOOL} \qquad \Gamma \vdash e_1 \,:\, t \qquad \Gamma \vdash e_2 \,:\, t}{\Gamma \vdash \text{IF } b \text{ THEN } e_1 \text{ ELSE } e_2 \,:\, t}\;\text{ET\_IF}$$

$$\frac{\Gamma \vdash e_1 \,:\, t_1 \qquad t_1 \bullet \Gamma \vdash e_2 \,:\, t_2}{\Gamma \vdash \text{LET } e_1 \text{ IN } e_2 \,:\, t_2}\;\text{ET\_LET}$$

Figure 4: The typing rules for source language expressions

Distribution	Parameter type	Domain	Distr. type	Density function
Bernoulli	REAL	$p \in [0;1]$	BOOL	$\begin{cases} p & \text{for } x = \text{TRUE} \\ 1-p & \text{for } x = \text{FALSE} \end{cases}$
UniformInt	INTEG × INTEG	$p_1 \leq p_2$	INTEG	$\langle x \in [p_1; p_2] \rangle$
UniformReal	REAL × REAL	$p_1 < p_2$	REAL	$\langle x \in [p_1; p_2] \rangle$
Gaussian	REAL × REAL	$p_2 > 0$	REAL	$\exp\left(-\frac{(x-p_1)^2}{2p_2^2}\right) \Big/ \sqrt{2\pi p_2^2}$
Poisson	REAL	$p \geq 0$	INTEG	$\begin{cases} \exp(-p) \cdot p^x / x! & \text{for } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$

Figure 5: The builtin distributions of the source language.
The density functions are given in terms of the parameter p, which is of the type given in the column "parameter type". If p is of a product type, p_1 and p_2 stand for the two components of p. x is the function variable, e. g. the point at which the density function is to be evaluated.

```
primrec expr_sem :: state ⇒ expr ⇒ val measure where
  expr_sem σ (Val v) = return_val v
| expr_sem σ (Var x) = return_val (σ x)
| expr_sem σ (LET e1 IN e2) =
      do {
        v ← expr_sem σ e1;
        expr_sem (v • σ) e2
      }
| expr_sem σ (op $ e) =
      do {
        v ← expr_sem σ e;
        return_val (op_sem op v)
      }
| expr_sem σ <e1, e2> =
      do {
        v ← expr_sem σ e1;
        w ← expr_sem σ e2;
        return_val <|v, w|>
      }
| expr_sem σ (IF b THEN e1 ELSE e2) =
      do {
        b′ ← expr_sem σ b;
        if b′ = TRUE then expr_sem σ e1 else expr_sem σ e2
      }
| expr_sem σ (Random dst e) =
      do {
        p ← expr_sem σ e;
        dist_measure dst p
      }
| expr_sem σ (Fail t) = null_measure (stock_measure t)
```

Figure 6: The semantics of source language expressions

3.4 Randomfree expressions

We call an expression *randomfree* if it contains no occurrence of Random or Fail. Such expressions are of particular interest as they are, in some sense, deterministic. If all their free variables have a fixed value, they return precisely one value, so we can define a function expr_sem_rf that, when given a state σ and a randomfree expression e, returns this single value.

The definition is obvious and leads to the following equality (assuming that e is randomfree and well-typed and σ is a valid state):

$$\mathrm{expr\_sem}\ \sigma\ e \ = \ \mathrm{return}\ (\mathrm{expr\_sem\_rf}\ \sigma\ e)$$

This property will later enable us to also convert randomfree source language expressions into "equivalent" target language expressions.

3.5 Target language

The target language is again modelled very closely after the one used by Bhat et al. [BBGR13]. The type system and the operators are the same as in the source language, but the key difference is that while expressions in the source language return a measure space on their result type, the expressions in the target language always return a single value.

Since our source language lacks sum types, so does our target language, of course; additionally, our target language differs from that used by Bhat et al. in the following respects:

- Our language has no function types; since functions only occur as integrands and as final results (as the compilation result is a density function), we can simply define integration to introduce the integration variable as a bound variable and let the final result contain a single free variable with de Bruijn index 0, i. e. there is an implicit λ abstraction around the compilation result.

- Evaluation of expressions in our target language can never fail. In the language by Bhat et al., failure is used to handle undefined integrals; we instead use the convention of Isabelle's measure theory library, which returns 0 for integrals of non-integrable functions. This has the advantage of keeping the semantics simple, which makes proofs considerably easier.

- Our target language does not have Let bindings, since, in contrast to the source language, they would be semantically superfluous here. However, they are still useful in practice since they yield shorter expressions and can avoid multiple evaluation of the same term; they could be added with little effort.

Figures 7, 8, and 9 show the syntax, typing rules, and semantics of the target language.

$$
\begin{array}{l}
\textbf{datatype}\ \mathrm{cexpr}\ =\\
\qquad \mathrm{CVar}\ \mathit{nat} \mid \mathrm{CVal}\ \mathit{val} \mid \mathit{pdf\_operator}\ \$_{\mathrm{c}}\ \mathit{cexpr} \mid {<}\mathit{cexpr}, \mathit{cexpr}{>}_{\mathrm{c}} \mid\\
\qquad \mathrm{IF_c}\ \mathit{cexpr}\ \mathrm{THEN}\ \mathit{cexpr}\ \mathrm{ELSE}\ \mathit{cexpr} \mid \int_{\mathrm{c}} \mathit{cexpr}\, \partial \mathit{pdf\_type}
\end{array}
$$

Figure 7: The expressions of the target language

$$
\text{CET\_VAL}\quad \frac{}{\Gamma \vdash_{\mathrm{c}} \mathrm{CVal}\ v\ :\ \mathrm{val\_type}\ v}
\qquad\qquad
\text{CET\_VAR}\quad \frac{}{\Gamma \vdash_{\mathrm{c}} \mathrm{CVar}\ x\ :\ \Gamma\ x}
$$

$$
\text{CET\_OP}\quad \frac{\Gamma \vdash_{\mathrm{c}} e\ :\ t \qquad \mathrm{op\_type}\ \mathit{op}\ t = \mathrm{Some}\ t'}{\Gamma \vdash_{\mathrm{c}}\ \mathit{op}\ \$_{\mathrm{c}}\ e\ :\ t'}
\qquad\qquad
\text{CET\_PAIR}\quad \frac{\Gamma \vdash_{\mathrm{c}} e_1\ :\ t_1 \qquad \Gamma \vdash_{\mathrm{c}} e_2\ :\ t_2}{\Gamma \vdash_{\mathrm{c}} {<}e_1, e_2{>}_{\mathrm{c}}\ :\ t_1 \times t_2}
$$

$$
\text{CET\_IF}\quad \frac{\Gamma \vdash_{\mathrm{c}} b\ :\ \mathrm{BOOL} \qquad \Gamma \vdash_{\mathrm{c}} e_1\ :\ t \qquad \Gamma \vdash_{\mathrm{c}} e_2\ :\ t}{\Gamma \vdash_{\mathrm{c}} \mathrm{IF_c}\ b\ \mathrm{THEN}\ e_1\ \mathrm{ELSE}\ e_2\ :\ t}
\qquad\qquad
\text{CET\_INT}\quad \frac{t \bullet \Gamma \vdash_{\mathrm{c}} e\ :\ \mathrm{REAL}}{\Gamma \vdash_{\mathrm{c}} \int_{\mathrm{c}} e\, \partial t\ :\ \mathrm{REAL}}
$$

Figure 8: The typing rules for the target language

$$
\begin{array}{l}
\textbf{primrec}\ \mathrm{cexpr\_sem}\ ::\ \mathrm{state} \Rightarrow \mathrm{cexpr} \Rightarrow \mathrm{val}\ \textbf{where}\\
\ \ \mathrm{cexpr\_sem}\ \sigma\ (\mathrm{CVal}\ v)\ =\ v\\
\mid \mathrm{cexpr\_sem}\ \sigma\ (\mathrm{CVar}\ x)\ =\ \sigma\ x\\
\mid \mathrm{cexpr\_sem}\ \sigma\ \ {<}e_1, e_2{>}_{\mathrm{c}} = {<}|\mathrm{cexpr\_sem}\ \sigma\ e_1, \mathrm{cexpr\_sem}\ \sigma\ e_2|{>}\\
\mid \mathrm{cexpr\_sem}\ \sigma\ \mathit{op}\ \$_{\mathrm{c}}\ e\ =\ \mathrm{op\_sem}\ \mathit{op}\ (\mathrm{cexpr\_sem}\ \sigma\ e)\\
\mid \mathrm{cexpr\_sem}\ \sigma\ (\mathrm{IF_c}\ b\ \mathrm{THEN}\ e_1\ \mathrm{ELSE}\ e_2)\ =\\
\qquad (\textbf{if}\ \mathrm{cexpr\_sem}\ \sigma\ b = \mathrm{TRUE}\ \textbf{then}\ \mathrm{cexpr\_sem}\ \sigma\ e_1\ \textbf{else}\ \mathrm{cexpr\_sem}\ \sigma\ e_2)\\
\mid \mathrm{cexpr\_sem}\ \sigma\ (\int_{\mathrm{c}} e\, \partial t)\ =\\
\qquad \mathrm{RealVal}\ (\int x.\ \mathrm{extract\_real}\ (\mathrm{cexpr\_sem}\ (x \bullet \sigma)\ e)\, \partial \mathrm{stock\_measure}\ t)
\end{array}
$$

Figure 9: The semantics of the target language

In our compiler, we will often have to modify target language expressions and combine them to larger ones. This requires the definition of a number of auxiliary functions e. g. to modify the variables in an expression, substitute an expression for a variable, etc. Using de Bruijn indices saves us the trouble of having to handle variable capture, but it still requires some care when combining expressions: when entering the scope of an integral, a new bound variable is introduced and all other variables are incremented. Therefore, when substituting any expression into the scope of an integral, all variables must be shifted by 1.

Moreover, as mentioned above, our target language does not contain the notion of a *function*, i. e. no λ abstractions or function applications. However, the compiler returns a density *function*, so we need to emulate functions by representing them as expressions with a single free variable with de Bruijn index 0. Effectively, this is the same as implicitly adding a λ around the expression. The drawback of this is that the inability to use abstraction, application, and β reduction requires us to introduce additional auxiliary functions in order to manipulate such expressions, e. g. to compose two functions. For a full list of these functions and other auxiliary functions related to the target language, see section A.3 in the appendix.

The function expr_rf_to_cexpr, which will be used in some rules of the compiler that handle randomfree expressions, is of particular interest. We mentioned earlier that randomfree source language expressions are deterministic in some sense and can thus be converted to target language expressions.[6] This function now does precisely that. its definition is mostly obvious, apart from the LET case. Since our target language does not have a LET construct, the function must resolve LET bindings in the source language expression by substituting the bound expression, which is done with the auxiliary function cexpr_subst.

expr_rf_to_cexpr satisfies the following equality for any randomfree source language expression e:

$$\text{cexpr\_sem}\ \sigma\ (\text{expr\_rf\_to\_cexpr}\ e)\ =\ \text{expr\_sem\_rf}\ \sigma\ e$$

[6]Bhat et al. say that a randomfree expression "is also an expression in the target language syntax, and we silently treat it as such" [BBGR13]

4 Abstract compiler

4.1 Density contexts

First, we define the notion of a *density context*, which holds the acquired context data the compiler will require to compute the density of an expression. A density context is a tuple $\Upsilon = (V, V', \Gamma, \delta)$ that contains the following information:

- The set V of random variables in the current context. These are the variables that are randomised over.
- The set V' of parameter variables in the current context. These are variables that may occur in the expression, but are not randomised over but treated as constants.
- The type environment Γ
- A density function δ that returns the common density of the variables V under the parameters V'. Here, δ is a function from space (state_measure $(V \cup V')$ Γ) to the extended real numbers.

A density context (V, V', Γ, δ) describes a parametrised measure on the states on $V \cup V'$. Let $\rho \in$ space (state_measure V' Γ) be a set of parameters. Then we write

$$\text{dens\_ctxt\_measure}\ (V, V', \Gamma, \delta)\ \rho$$

for the measure that we obtain by taking state_measure V Γ, transforming it by merging a given state σ with the parameter state ρ and finally applying the density δ on the resulting image measure. The Isabelle definition of this is:

definition dens_ctxt_measure :: dens_ctxt $\Rightarrow$ state $\Rightarrow$ state_measure **where**
    dens_ctxt_measure (V, V', Γ, δ) ρ =
        density (distr (state_measure V Γ) (state_measure $(V \cup V')$ Γ) $(\lambda\sigma.$ merge V V' $(\sigma, \rho))$) δ

A density context is *well-formed* (implemented in Isabelle as the locale density_context) if:

- V and V' are finite and disjoint
- $\delta\ \sigma \geq 0$ for any $\sigma \in$ space (state_measure $(V \cup V')$ Γ)
- δ is Borel-measurable w. r. t. state_measure $(V \cup V')$ Γ
- for any $\rho \in$ space (state_measure V' Γ), the measure dens_ctxt_measure (V, V', Γ, δ) ρ is a subprobability measure

4.2 Definition

As a first step, we have implemented an abstract density compiler as an inductive predicate $\Upsilon \vdash_{\mathrm{d}} e \Rightarrow f$, where Υ is a density context, e is a source language expression and f is a function of type val state $\Rightarrow$ val $\Rightarrow$ ereal. Its first parameter is a state that assigns values to the free variables in e and its second parameter is the value for which the density is to be computed. The compiler therefore computes a density function that is parametrised with the values of the non-random free variables in the source expression.

The compilation rules are virtually identical to those by Bhat et al. [BBGR13], except for the following adaptations:

- Bhat et al. handle the IF ... THEN ... ELSE case with the "match" rule for sum types. Since we do not support sum types, we have a dedicated rule for IF ... THEN ... ELSE.
- The use of de Bruijn indices requires shifting of variable sets and states whenever the scope of a new bound variable is entered; unfortunately, this makes some rules somewhat technical.
- We do not provide any compiler support for *deterministic* LETs. They are semantically redundant, as they can always be expanded without changing the semantics of the expression. In fact, they *have* to be unfolded for compilation, so they can be regarded as a feature that adds convenience, but no expressivity.

The following list shows the standard compilation rules adapted from Bhat et al., plus a rule for multiplication with a constant. Note that the functions marg_dens and marg_dens2 compute the marginal density of one (resp. two) variables by "integrating away" all the other variables from the common density δ. The function branch_prob computes the probability of being in the current branch of execution by integrating over *all* the variables in the common density δ.

HD_VAL

$$\frac{\text{countable\_type}\ (\text{val\_type}\ v)}{(V, V', \Gamma, \delta) \vdash_{\mathrm{d}} \text{Val}\ v \Rightarrow \lambda\rho\, x.\ \text{branch\_prob}\ (V, V'\Gamma, \delta)\ \rho \cdot \langle x = v\rangle}$$

HD_VAR

$$\frac{x \in V}{(V, V', \Gamma, \delta) \vdash_{\mathrm{d}} \text{Var}\ x \Rightarrow \text{marg\_dens}\ (V, V', \Gamma, \delta)\ x}$$

HD_PAIR

$$\frac{x \in V \qquad y \in V \qquad x \neq y}{(V, V', \Gamma, \delta) \vdash_{\mathrm{d}} <\text{Var}\ x, \text{Var}\ y> \Rightarrow \text{marg\_dens2}\ (V, V', \Gamma, \delta)\ x\ y}$$

HD_FAIL

$$\frac{}{(V, V', \Gamma, \delta) \vdash_{\mathrm{d}} \text{Fail}\ t \Rightarrow \lambda\rho\, x.\ 0}$$

HD_LET

$$\frac{\begin{array}{c}(\emptyset, V \cup V', \Gamma, \lambda x.\ 1) \vdash_{\mathrm{d}} e_1 \Rightarrow f \\ (0 \bullet V, \{x+1 \mid x \in V'\}, \text{type\_of}\ \Gamma\ e_1 \bullet \Gamma, (f \cdot \delta)(\rho \circ (\lambda x.\ x+1)) \vdash_{\mathrm{d}} e_2 \Rightarrow g\end{array}}{(V, V', \Gamma, \delta) \vdash_{\mathrm{d}} \text{LET}\ e_1\ \text{IN}\ e_2 \Rightarrow \lambda\rho.\ g\ (\text{undefined} \bullet \rho)}$$

HD_RAND

$$\frac{(V, V', \Gamma, \delta) \vdash_{\mathrm{d}} e \Rightarrow f}{(V, V', \Gamma, \delta) \vdash_{\mathrm{d}} \text{Random } \mathit{dst}\ e \Rightarrow \lambda\rho\, x.\ \int^{+} x.\ f\ \rho\ x \cdot \text{dist\_dens } \mathit{dst}\ x\ y\ \partial \text{dist\_param\_type } \mathit{dst}}$$

HD_RAND_DET

$$\frac{\text{randomfree } e \qquad \text{free\_vars } e \subseteq V'}{\begin{array}{l}(V, V', \Gamma, \delta) \vdash_{\mathrm{d}} \text{Random } \mathit{dst}\ e \Rightarrow \\ \qquad \lambda\rho\, x.\ \text{branch\_prob } (V, V', \Gamma, \delta)\ \rho \cdot \text{dist\_dens } \mathit{dst}\ (\text{expr\_sem\_rf } \rho\ e)\ x\end{array}}$$

HD_IF

$$\frac{\begin{array}{c}(\emptyset, V \cup V', \Gamma, \lambda\rho.\ 1) \vdash_{\mathrm{d}} b \Rightarrow f \\ (V, V', \Gamma, \lambda\rho.\ \delta\ \rho \cdot f\ \text{TRUE}) \vdash_{\mathrm{d}} e_1 \Rightarrow g_1 \qquad (V, V', \Gamma, \lambda\rho.\ \delta\ \rho \cdot f\ \text{FALSE}) \vdash_{\mathrm{d}} e_2 \Rightarrow g_2\end{array}}{(V, V', \Gamma, \delta) \vdash_{\mathrm{d}} \text{IF } b \text{ THEN } e_1 \text{ ELSE } e_2 \Rightarrow \lambda\rho\, x.\ g_1\ \rho\ x + g_2\ \rho\ x}$$

HD_IF_DET

$$\frac{\begin{array}{c}\text{randomfree } b \\ (V, V', \Gamma, \lambda\rho.\ \delta\ \rho \cdot \langle \text{expr\_sem\_rf } \rho\ b = \text{TRUE} \rangle) \vdash_{\mathrm{d}} e_1 \Rightarrow g_1 \\ (V, V', \Gamma, \lambda\rho.\ \delta\ \rho \cdot \langle \text{expr\_sem\_rf } \rho\ b = \text{FALSE} \rangle) \vdash_{\mathrm{d}} e_2 \Rightarrow g_2\end{array}}{(V, V', \Gamma, \delta) \vdash_{\mathrm{d}} \text{IF } b \text{ THEN } e_1 \text{ ELSE } e_2 \Rightarrow \lambda\rho\, x.\ g_1\ \rho\ x + g_2\ \rho\ x}$$

HD_FST

$$\frac{(V, V', \Gamma, \delta) \vdash_{\mathrm{d}} e \Rightarrow f}{(V, V', \Gamma, \delta) \vdash_{\mathrm{d}} \text{fst } e \Rightarrow \lambda\rho\, x.\ \int^{+} y.\ f\ \rho\ <|x, y|>\ \partial \text{type\_of } \Gamma\ (\text{snd } e)}$$

HD_SND

$$\frac{(V, V', \Gamma, \delta) \vdash_{\mathrm{d}} e \Rightarrow f}{(V, V', \Gamma, \delta) \vdash_{\mathrm{d}} \text{snd } e \Rightarrow \lambda\rho\, y.\ \int^{+} x.\ f\ \rho\ <|x, y|>\ \partial \text{type\_of } \Gamma\ (\text{fst } e)}$$

HD_OP_DISCR

$$\frac{\text{countable\_type } (\text{type\_of } (\mathit{op}\,\$\, e)) \qquad (V, V', \Gamma, \delta) \vdash_{\mathrm{d}} e \Rightarrow f}{(V, V', \Gamma, \delta) \vdash_{\mathrm{d}} \mathit{op}\,\$\, e \Rightarrow \lambda\rho\, y.\ \int^{+} x.\ \langle \text{op\_sem } \mathit{op}\ x = y \rangle \cdot f\ \rho\ x\, \partial \text{type\_of } \Gamma\ e}$$

HD_NEG

$$\frac{(V, V', \Gamma, \delta) \vdash_{\mathrm{d}} e \Rightarrow f}{(V, V', \Gamma, \delta) \vdash_{\mathrm{d}} -e \Rightarrow \lambda\rho\, x.\ f\ \rho\ (-x)}$$

HD_ADDC

$$\frac{\text{randomfree } e' \qquad \text{free\_vars } e' \subseteq V' \qquad (V, V', \Gamma, \delta) \vdash_{\mathrm{d}} e \Rightarrow f}{(V, V', \Gamma, \delta) \vdash_{\mathrm{d}} e + e' \Rightarrow \lambda\rho\, x.\ f\ \rho\ (x - \text{expr\_sem\_rf } \rho\ e')}$$

HD_MULTC

$$\frac{c \neq 0 \qquad (V, V', \Gamma, \delta) \vdash_{\mathrm{d}} e \Rightarrow f}{(V, V', \Gamma, \delta) \vdash_{\mathrm{d}} e \cdot \text{Val } (\text{RealVal } c) \Rightarrow \lambda\rho\, x.\ f\ \rho\ (x/c)\ /\ |c|}$$

HD_ADD

$$\frac{(V, V', \Gamma, \delta) \vdash_{\mathrm{d}} <e_1, e_2> \Rightarrow f}{(V, V', \Gamma, \delta) \vdash_{\mathrm{d}} e_1 + e_2 \Rightarrow \lambda\rho\, z.\ \int^{+} x.\ f\ \rho\ <|x, z - x|>}$$

HD_INV

$$\frac{(V, V', \Gamma, \delta) \vdash_{\mathrm{d}} e \Rightarrow f}{(V, V', \Gamma, \delta) \vdash_{\mathrm{d}} e^{-1} \Rightarrow \lambda \rho\, x.\ f\ \rho\ (x^{-1}) \,/\, x^2}$$

HD_EXP

$$\frac{(V, V', \Gamma, \delta) \vdash_{\mathrm{d}} e \Rightarrow f}{(V, V', \Gamma, \delta) \vdash_{\mathrm{d}} \exp e \Rightarrow \lambda \rho\, x.\ \textbf{if}\ x > 0\ \textbf{then}\ f\ \rho\ (\ln x) \,/\, x\ \textbf{else}\ 0}$$

Figure 10: The abstract compilation rules

There is a slight technical problem, namely that functions in Isabelle must be total. Since the returned density functions have type val state $\Rightarrow$ val $\Rightarrow$ ereal, they must also be defined for values outside their domain. E. g. a density function for type REAL must also return some density for an input value of type BOOL. In the above formalisation, this will be some unspecified value; the density computed by the concrete compiler may return some other value outside its domain. The same is true for the common density δ in the density context. Additionally the concrete density function may deviate from the abstract one on a null set, i. e. it is enough if they are equal almost everywhere.

To account for this, we have three additional rules, which are too technical to be printed here:

HD_AE states that if $\Upsilon \vdash_{\mathrm{d}} e \Rightarrow f$ and f' is measurable and for any parameter state ρ, the function $f'\ \rho$ is nonnegative and equal to $f\ \rho$ almost everywhere, then $\Upsilon \vdash_{\mathrm{d}} e \Rightarrow f'$ also holds.

HD_DENS_CTXT_CONG states that if $(V, V', \Gamma, \delta) \vdash_{\mathrm{d}} e \Rightarrow f$ and δ is equal to some δ' for all states on $V \cup V'$, then $(V, V', \Gamma, \delta') \vdash_{\mathrm{d}} e \Rightarrow f$ also holds.

HD_CONG is derived from HD_AE; it assumes that f and f' are equal on their entire domain and in turn removes the measurability and nonnegativity assumptions.

4.3 Soundness proof

We show the following soundness result for the abstract compiler: [7]

lemma expr_has_density_sound :
  assumes $(\emptyset, \emptyset, \Gamma, \lambda\rho.\, 1) \vdash_{\mathrm{d}} e \Rightarrow f$ **and** $\Gamma \vdash e : t$ **and** free_vars $e = \emptyset$
  shows has_subprob_density (expr_sem σ e) (stock_measure t) (f ($\lambda x.$ undefined))

Here, has_subprob_density M N f is an abbreviation for the following four facts:

- applying the density f to N yields M
- M is a subprobability measure
- f is N-Borel-measurable
- f is nonnegative on its domain

We prove this with the following generalised auxiliary lemma:

lemma expr_has_density_sound_aux :
  assumes $(V, V', \Gamma, \delta) \vdash_{\mathrm{d}} e \Rightarrow f$ **and** $\Gamma \vdash e : t$ **and** free_vars $e = \emptyset$ **and**
  density_context V V' Γ δ **and** free_vars $e \subseteq V \cup V'$
  shows has_parametrized_subprob_density (state_measure V' Γ)
    ($\lambda\rho.$ **do** $\{\sigma \leftarrow$ dens_ctxt_measure (V, V', Γ, δ) ρ; expr_sem σ $e\}$)
    (stock_measure t) f

Here, the predicate has_parametrized_subprob_density R M N f simply means that f is Borel-measurable w. r. t. $R \otimes N$ and that for any parameter state ρ from R, the predicate has_subprob_density M N f holds.

The proof is a straightforward induction following the inductive definition of the abstract compiler. In many cases, the monad laws for the Giry monad allow restructuring the induction goal in such a way that the induction hypothesis can be applied directly; in the other cases, the definitions need to be unfolded and the goal is essentially to show that two integrals are equal and that the output produced is well-formed.

The proof given by Bhat et al. [BBGR14] is analogous to ours, but much more concise due to the fact that side conditions such as measurability, integrability, nonnegativity, and so on are not proven explicitly and many important steps are skipped or only hinted at.

[7] Note that since the abstract compiler returns parametrised density functions, we need to parametrise the result with the state $\lambda x.$ undefined, even if the expression contains no free variables.

5 Concrete compiler

5.1 Approach

The concrete compiler is another inductive predicate, modelled directly after the abstract compiler, but returning a target language expression as the compilation result instead of a HOL function. We will use a standard refinement approach to relate the concrete compiler to the abstract compiler and thus lift the soundness result on the abstract compiler to an analogous one on the concrete one, effectively showing that the concrete compiler always returns a well-formed target language expression that represents a density for the subprobability space described by the source language.

The concrete compilation predicate is written as

$$(vs, vs', \Gamma, \delta) \vdash_c e \Rightarrow f$$

Here, vs and vs' are lists of variables, Γ is a typing environment, and δ is a target-language expression describing the common density of the random variables vs in the context. It may be parametrised with the variables from vs'.

5.2 Definition

The concrete compilation rules are, of course, a direct copy of the abstract ones, but with all the abstract HOL operations replaced with operations on target language expressions. Due to the de Bruijn indices and the lack of functions as explicit objects in the target language, some of the rules are somewhat complicated, as inserting an expression into the scope of one or more bound variables (such as under an integral) requires shifting the variable indices of the inserted expression correctly.

The following list shows these rules. They heavily use the auxiliary functions listed in section A.3 in the appendix.

EDC_VAL

$$\frac{\text{countable\_type}\ (\text{val\_type}\ v)}{(vs, vs', \Gamma, \delta) \vdash_c \text{Val}\ v \Rightarrow \text{map\_vars Suc}\ (\text{branch\_prob\_cexpr}\ (vs, vs', \Gamma, \delta)) \cdot \langle \text{CVar}\ 0\ =\ \text{CVal}\ v \rangle}$$

EDC_VAR

$$\frac{x \in \text{set}\ vs}{(vs, vs', \Gamma, \delta) \vdash_c \text{Var}\ x \Rightarrow \text{marg\_dens\_cexpr}\ \Gamma\ vs\ x\ \delta}$$

EDC_PAIR

$$\frac{x \in \text{set}\ vs \qquad y \in \text{set}\ vs}{(vs, vs', \Gamma, \delta) \vdash_c <x, y> \Rightarrow \text{marg\_dens2\_cexpr}\ \Gamma\ \text{vs}\ x\ y\ \delta}$$

EDC_FAIL

$$\frac{}{(vs, vs', \Gamma, \delta) \vdash_c \text{Fail}\ t \Rightarrow \text{CVal}\ (\text{RealVal}\ 0)}$$

EDC_LET

$$\frac{([], vs\ @\ vs', \Gamma, 1) \vdash_c e \Rightarrow f \qquad (0\ \#\ \text{map Suc}\ vs,\ \text{map Suc}\ vs',\ \text{type\_of}\ \Gamma\ e \bullet \Gamma,\ \text{map\_vars Suc}\ \delta \cdot f) \vdash_c e' \Rightarrow g}{(vs, vs', \Gamma, \delta) \vdash_c \text{LET}\ e\ \text{IN}\ e' \Rightarrow \text{map\_vars}\ (\lambda x.\ x - 1)\ g}$$

EDC_RAND

$$\frac{(vs, vs', \Gamma, \delta) \vdash_{\mathrm{c}} e \;\Rightarrow f}{\begin{array}{l}(vs, vs', \Gamma, \delta) \vdash_{\mathrm{c}} \text{Random } dst\ e \;\Rightarrow \\ \qquad \int_c \text{map\_vars } (\text{case\_nat } 0\ (\lambda x.\ x+2))\ f \cdot \\ \qquad\quad \text{dist\_dens\_cexpr } dst\ (\text{CVar } 0)\ (\text{CVar } 1)\ \partial\, \text{dist\_param\_type } dst\end{array}}$$

EDC_RAND_DET

$$\frac{\text{randomfree } e \qquad \text{free\_vars } e \subseteq \text{set } vs'}{\begin{array}{l}(vs, vs', \Gamma, \delta) \vdash_{\mathrm{c}} \text{Random } dst\ e \;\Rightarrow \\ \qquad \text{map\_vars Suc } (\text{branch\_prob\_cexpr } (vs, vs', \Gamma, \delta)) \cdot \\ \qquad \text{dist\_dens\_cexpr } dst\ (\text{map\_vars Suc } (\text{expr\_rf\_to\_cexpr e}))\ (\text{CVar } 0)\end{array}}$$

EDC_IF

$$\frac{\begin{array}{c}([], vs\ @\ vs', \Gamma, 1) \vdash_{\mathrm{c}} b \;\Rightarrow f \\ (vs, vs', \Gamma, \delta \cdot \langle \text{cexpr\_subst\_val } f \text{ TRUE}\rangle) \vdash_{\mathrm{c}} e_1 \;\Rightarrow f_1 \\ (vs, vs', \Gamma, \delta \cdot \langle \text{cexpr\_subst\_val } f \text{ FALSE}\rangle) \vdash_{\mathrm{c}} e_2 \;\Rightarrow f_2\end{array}}{(vs, vs', \Gamma, \delta) \vdash_{\mathrm{c}} \text{IF } b \text{ THEN } e_1 \text{ ELSE } e_2 \;\Rightarrow f_1 + f_2}$$

EDC_IF_DET

$$\frac{\begin{array}{c}\text{randomfree } b \\ (vs, vs', \Gamma, \delta \cdot \langle \text{expr\_rf\_to\_cexpr } b\rangle) \vdash_{\mathrm{c}} e_1 \;\Rightarrow f_1 \\ (vs, vs', \Gamma, \delta \cdot \langle \neg\, \text{expr\_rf\_to\_cexpr } b\rangle) \vdash_{\mathrm{c}} e_2 \;\Rightarrow f_2\end{array}}{(vs, vs', \Gamma, \delta) \vdash_{\mathrm{c}} \text{IF } b \text{ THEN } e_1 \text{ ELSE } e_2 \;\Rightarrow f_1 + f_2}$$

EDC_OP_DISCR

$$\frac{\Gamma \vdash e : t \qquad \text{op\_type } oper\ t = \text{Some } t' \qquad \text{countable\_type } t' \qquad (vs, vs', \Gamma, \delta) \vdash_{\mathrm{c}} e \Rightarrow f}{(vs, vs', \Gamma, \delta) \vdash_{\mathrm{c}} oper\ \$\ e \;\Rightarrow \int_c \langle oper\ \$_{\mathrm{c}}\ \text{CVar } 0 \;=\; \text{CVar } 1\rangle \cdot \text{map\_vars } (\text{case\_nat } 0\ (\lambda x.\ x+2))\ f\ \partial t}$$

EDC_FST

$$\frac{\Gamma \vdash e : t \times t' \qquad (vs, vs', \Gamma, \delta) \vdash_{\mathrm{c}} e \Rightarrow f}{(vs, vs', \Gamma, \delta) \vdash_{\mathrm{c}} \text{fst } e \;\Rightarrow \int_c \text{map\_vars } (\text{case\_nat } 0\ (\lambda x.\ x+2))\ f\ \circ_c <\text{CVar } 1, \text{CVar } 0>_{\mathrm{c}}\ \partial t'}$$

EDC_SND

$$\frac{\Gamma \vdash e : t \times t' \qquad (vs, vs', \Gamma, \delta) \vdash_{\mathrm{c}} e \Rightarrow f}{(vs, vs', \Gamma, \delta) \vdash_{\mathrm{c}} \text{snd } e \;\Rightarrow \int_c \text{map\_vars } (\text{case\_nat } 0\ (\lambda x.\ x+2))\ f\ \circ_c <\text{CVar } 0, \text{CVar } 1>_{\mathrm{c}}\ \partial t}$$

EDC_NEG

$$\frac{(vs, vs', \Gamma, \delta) \vdash_{\mathrm{c}} e \Rightarrow f}{(vs, vs', \Gamma, \delta) \vdash_{\mathrm{c}} -e \Rightarrow f\ \circ_c\ (-\text{CVar } 0)}$$

EDC_ADDC

$$\frac{(vs, vs', \Gamma, \delta) \vdash_{\mathrm{c}} e \Rightarrow f \qquad \text{randomfree } e' \qquad \text{free\_vars } e' \subseteq \text{set } vs'}{(vs, vs', \Gamma, \delta) \vdash_{\mathrm{c}} e + e' \;\Rightarrow f\ \circ_c\ (\text{CVar } 0 - \text{map\_vars Suc } (\text{expr\_rf\_to\_cexpr } e'))}$$

EDC_MULTC

$$\frac{(vs, vs', \Gamma, \delta) \vdash_{\mathrm{c}} e \Rightarrow f \qquad c \neq 0}{(vs, vs', \Gamma, \delta) \vdash_{\mathrm{c}} e \cdot \text{Val } (\text{RealVal } c) \;\Rightarrow (f \circ_c (\text{CVar } 0\ /\ c))/|c|}$$

EDC_ADD

$$\frac{\Gamma \vdash <e, e'> : t \times t \qquad (vs, vs', \Gamma, \delta) \vdash_{\mathrm{c}} <e, e'> \Rightarrow f}{(vs, vs', \Gamma, \delta) \vdash_{\mathrm{c}} e + e' \;\Rightarrow \int_c \text{map\_vars } (\text{case\_nat } 0\ (\lambda x.\ x+2))\ f \circ_c <\text{CVar } 0, \text{CVar } 1 - \text{CVar } 0>\ \partial t}$$

EDC_INV

$$\frac{(vs, vs', \Gamma, \delta) \vdash_c e \Rightarrow f}{(vs, vs', \Gamma, \delta) \vdash_c e^{-1} \Rightarrow (f \circ_c (1/\mathrm{CVar}\ 0))\ /\ (\mathrm{CVar}\ 0)^2}$$

EDC_EXP

$$\frac{(vs, vs', \Gamma, \delta) \vdash_c e \Rightarrow f}{(vs, vs', \Gamma, \delta) \vdash_c \exp e \Rightarrow \mathrm{IF}\ \ \mathrm{CVar}\ 0 > 0\ \ \mathrm{THEN}\ (f \circ_c \ln (\mathrm{CVar}\ 0))\ /\ \mathrm{CVar}\ 0\ \mathrm{ELSE}\ 0}$$

Figure 11: The concrete compilation rules

5.3 Refinement

The refinement relates the concrete compilation

$$(vs, vs', \Gamma, \delta) \vdash_c e \Rightarrow f$$

to the abstract compilation

$$(\mathrm{set}\ vs, \mathrm{set}\ vs', \Gamma, \lambda\sigma.\ \mathrm{cexpr\_sem}\ \sigma\ \delta) \vdash_c e \Rightarrow \lambda\rho\, x.\ \mathrm{cexpr\_sem}\ (x \bullet \rho)\ f$$

In words: we take the abstract compilation predicate and

- variable sets are refined to variable lists
- the typing context and the source language expression remain unchanged
- the common density in the context and the compilation result are refined from HOL functions to target language expressions (by applying the target language semantics)

The main refinement lemma states that the concrete compiler yields a result that is equivalent to that of the abstract compiler, modulo refinement. Informally, the statement is the following: if e is ground and well-typed under some well-formed concrete density context Υ and $\Upsilon \vdash_c e \Rightarrow f$, then $\Upsilon' \vdash_d e \Rightarrow f'$, where Υ' and f' are the abstract versions of Υ and f.

The proof for this is conceptually simple – induction over the definition of the concrete compiler; in practice, however, it is quite involved. In every single induction step, the wellformedness of the intermediary expressions needs to be shown, the congruence lemmas for the abstract compiler need to be applied (see section 4.2), and, when integration is involved, nonnegativity and integrability have to be shown in order to convert nonnegative integrals to Lebesgue integrals and integrals on product spaces to iterated integrals.

Combining this main refinement lemma and the abstract soundness lemma, we can now easily show the concrete soundness lemma:

$$
\begin{array}{l}
\textbf{lemma}\ \text{expr\_has\_density\_cexpr\_sound}: \\
\quad \textbf{assumes}\ \ ([], [], \Gamma, 1) \vdash_{\mathrm{c}} e \Rightarrow f \quad \textbf{and} \quad \Gamma \vdash e : t \quad \textbf{and} \quad \text{free\_vars}\ e = \emptyset \\
\quad \textbf{shows}\ \ \text{has\_subprob\_density}\ (\text{expr\_sem}\ \sigma\ e)\ (\text{stock\_measure}\ t) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\lambda x.\ \text{cexpr\_sem}\ (x \bullet \sigma)\ f) \\
\qquad\qquad\ \ \Gamma'\ 0 = t \Longrightarrow \Gamma' \vdash_{\mathrm{c}} f : \text{REAL} \\
\qquad\qquad\ \ \text{free\_vars}\ f \subseteq \{0\}
\end{array}
$$

Informally, the lemma states that if e is a well-typed, ground source-language expression, compiling it with an empty context will yield a well-typed, well-formed target language expression representing a density function on the measure space described by e.

5.4 Final result

For convenience, we define the symbol $e : t \Rightarrow_{\mathrm{c}} f$ (read "e with type t compiles to f"), which includes the well-typedness and groundness requirements on e as well as the compilation result: [8]

$$
e : t \Rightarrow_{\mathrm{c}} f \quad \longleftrightarrow \quad (\lambda x.\ \text{UNIT} \vdash e : t \ \wedge\ \text{free\_vars}\ e = \emptyset \ \wedge\ ([], [], \lambda x.\ \text{UNIT}, 1) \vdash_{\mathrm{c}} e \Rightarrow f)
$$

The final soundness theorem for the compiler is then:[9]

$$
\begin{array}{l}
\textbf{lemma}\ \text{expr\_compiles\_to\_sound}: \\
\quad \textbf{assumes}\ \ e : t \Rightarrow_{\mathrm{c}} f \\
\quad \textbf{shows}\ \ \ \ \ \text{expr\_sem}\ \sigma\ e = \text{density}\ (\text{stock\_measure}\ t)\ (\lambda x.\ \text{cexpr\_sem}\ (x \bullet \sigma')\ f) \\
\qquad\qquad\qquad \forall x \in \text{type\_universe}\ t.\ \text{cexpr\_sem}\ (x \bullet \sigma')\ f \geq 0 \\
\qquad\qquad\qquad \Gamma \vdash e : t \\
\qquad\qquad\qquad t \bullet \Gamma' \vdash_{\mathrm{c}} f : \text{REAL} \\
\qquad\qquad\qquad \text{free\_vars}\ f \subseteq \{0\}
\end{array}
$$

[8] In this definition, the choice of the typing environment is, of course, completely arbitrary since the expression contains no free variables.

[9] To be precise, the lemma statement in Isabelle is slightly different; for better readability, we unfolded one auxiliary definition here and omitted the type cast from real to ereal.

In words, this result means the following:

Theorem

Let e be a source language expression. If the compiler determines that e is well-formed and well-typed with type t and returns the target language expression f, then:

- the measure obtained by taking the stock measure of t and using the evaluation of f as a density is precisely the measure obtained by evaluating e
- f is nonnegative on all input values of type t
- e has no free variables and indeed has type t (in any type environment Γ)
- f has no free variable except the parameter (i. e. the variable 0) and is a function from t to REAL[10]

Isabelle's code generator now allows us to execute our inductively-defined verified compiler using the **values** command[11] or generate code in one of the target languages such as Standard ML or Haskell.

5.5 Evaluation

As an example on which to test the compiler, we choose the same expression that was chosen by Bhat et al. [BBGR13][12]:

$$\begin{array}{l}\textbf{LET}\ \text{Random UniformReal}\ {<}0,1{>}\ \textbf{IN}\\ \quad\textbf{LET}\ \text{Random Bernoulli}\ (\text{Var}\ 0)\ \textbf{IN}\\ \quad\quad\textbf{IF}\ \text{Var}\ 0\ \textbf{THEN}\ \text{Var}\ 1\ +\ 1\ \textbf{ELSE}\ \text{Var}\ 1\end{array}$$

Using symbolic variable names instead of de Bruijn indices, the expression would look like this:

$$\begin{array}{l}\textbf{LET}\ x = \text{Random UniformReal}\ {<}0,1{>}\ \textbf{IN}\\ \quad\textbf{LET}\ y = \text{Random Bernoulli}\ x\ \textbf{IN}\\ \quad\quad\textbf{IF}\ y\ \textbf{THEN}\ x\ +\ 1\ \textbf{ELSE}\ x\end{array}$$

We abbreviate this expression with e. We can then display the result of the compilation using the following Isabelle command:

$$\textbf{values}\ ''\{(t,f)\mid t\ f.\ e : t \Rightarrow_{\mathrm{c}} f\}''$$

[10] meaning if its parameter variable has type t, it is of type REAL

[11] Our compiler is inherently nondeterministic since it may return zero, one, or many density functions, seeing as an expression may have no matching compilation rules or more than one. Therefore, we must use the **values** command instead of the **value** command and receive a set of compilation results.

[12] Val and RealVal were omitted for better readability

The result is a singleton set which contains the pair (REAL, f), where f is a very long and complicated expression. Simplifying constant subexpressions and expressions of the form fst $<e_1, e_2>$ and again replacing de Bruijn indices with symbolic identifiers, we obtain:

$$\begin{aligned}
&\textstyle\int b.\ (\textbf{IF } 0 \leq x - 1 \wedge x - 1 \leq 1 \textbf{ THEN } 1 \textbf{ ELSE } 0)\cdot \\
&\qquad (\textbf{IF } 0 \leq x - 1 \wedge x - 1 \leq 1 \textbf{ THEN IF } b \textbf{ THEN } x - 1 \textbf{ ELSE } 1 - (x-1) \textbf{ ELSE } 0) \cdot \langle b \rangle + \\
&\textstyle\int b.\ (\textbf{IF } 0 \leq x \wedge x \leq 1 \textbf{ THEN } 1 \textbf{ ELSE } 0)\cdot \\
&\qquad (\textbf{IF } 0 \leq x \wedge x \leq 1 \textbf{ THEN IF } b \textbf{ THEN } x \textbf{ ELSE } 1 - x \textbf{ ELSE } 0) \cdot \langle \neg\, b \rangle
\end{aligned}$$

Further simplification yields the following result:

$$\langle 1 \leq x \leq 2 \rangle \cdot (x - 1) + \langle 0 \leq x \leq 1 \rangle \cdot (1 - x)$$

While this result is the same as that which Bhat et al. have reached, our compiler generates a much larger expression than the one they printed. The reason for this is that they printed a β-reduced version of the compiler output; in particular, constant subexpressions were evaluated. While such simplification is, of course, very useful when using the compiler in practice, we have not implemented it since it is not conceptually interesting and outside the scope of this work.

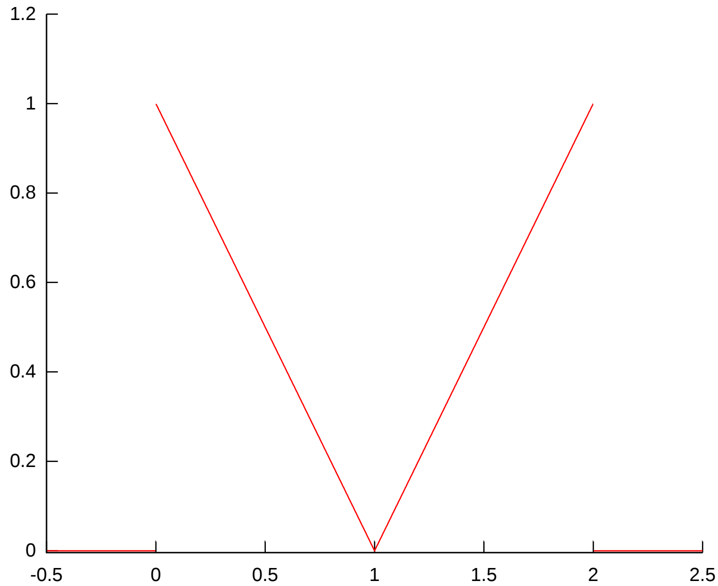

Figure 12: The graph of the density function of the example expression

6 Conclusion

6.1 Breakdown

All in all, the formalisation of the compiler took about three months. It contains a total of 12000 lines of Isabelle code (definitions, lemma statements, proofs, and examples). Figure 13 shows a detailed breakdown of this.

Type system and semantics	3400 lines
Abstract compiler	2800 lines
Concrete compiler	1400 lines
General Measure Theory / auxiliary lemmas	3300 lines

Figure 13: Breakdown of the Isabelle code

As can be seen from this figure, a sizeable portion of the work was the formalisation of results from general measure theory, such as the substitution lemmas for Lebesgue integrals, and auxiliary notions and lemmas, such as measure embeddings. Since the utility of these formalisations is not restricted to this particular project, they will be moved to Isabelle's Measure Theory library.

6.2 Difficulties

The main problems we encountered during the formalisation were:

Missing background theory

As mentioned in the previous section, a sizeable amount of Measure Theory and auxiliary notions had to be formalised. Most notably, the existing Measure Theory library did not contain integration by substitution. As a side product of their formalisation of the Central Limit Theorem [AHS14], Avigad et al. proved the Fundamental Theorem of Calculus and, building thereupon, integration by substitution. However, their integration-by-substitution lemma only supported continuous functions, whereas we required the theorem for general Borel-measurable functions. Using their proof of the Fundamental Theorem of Calculus, we proved such a lemma, which initially comprised almost 1000 lines of proof, but has been shortened significantly thereafter.

Proving side conditions

Many lemmas from the Measure Theory library require measurability, integrability, nonnegativity, etc. In hand-written proofs, this is often "handwaved" or implicitly dismissed as trivial; in a formal proof, proving these can blow up proofs and render them very complicated and technical. The measurability proofs in particular are ubiquitous in our formalisation. The Measure Theory library provides some tools for proving measurability automatically, but while they were quite helpful in many cases, they are still work in progress and require more tuning.

Lambda calculus

Bhat et al. use a simply-typed Lambda calculus-like language with symbolic identifiers as a target language. For a paper proof, this is the obvious choice, since it leads to concise and familiar definitions, but in a formal setting, it always comes with the typical problems of having to deal with variable capture and related issues. For that reason, we chose to use de Bruijn indices instead; however, this makes handling target language terms less intuitive, since variable indices need to be shifted whenever several target language terms are combined.

Another issue was the lack of a function type in our target language; allowing first-order functions, as Bhat et al. did, would have made many definitions easier and more natural, but would have complicated others significantly due to measurability issues.

Furthermore, we effectively needed to formalise a number of properties of Lambda calculus that can be implicitly used in a paper proof.

6.3 Future work

The following improvements to the Isabelle formalisation could probably be realised with little effort:

- sum types and a **match**… **with**… statement
- a compiler rule for deterministic let bindings
- let bindings for the target language
- a preprocessing stage to allow "normal" variables instead of de Bruijn indices
- a postprocessing stage to automatically simplify the density expression as far as possible

The first of these is interesting because it is the only substantial weakness of our formalisation as opposed to that by Bhat et al.; the remaining four merely make using the compiler more convenient or more efficient.

Additionally, in the long term, a *Markov chain Monte Carlo (MCMC)* sampling method such as the *Metropolis-Hastings algorithm* could also be formalised in Isabelle and integrated with the density compiler. However, this would be a more involved project as it will probably require the formalisation of additional probability theory in Isabelle.

6.4 Summary

Using Isabelle/HOL, we formalised the semantics of a simple probabilistic functional programming language with predefined probability distributions and a compiler that returns the probability distribution that a program in this language describes. These are modelled very closely after those given by Bhat et al. [BBGR13]. We then used the existing formalisations of measure theory in Isabelle/HOL to formally prove the correctness of this compiler w. r. t. the semantics of the source and target languages.

This shows not only that the compiler given by Bhat et al. is correct, but also that a formal correctness proof for such a compiler can be done with reasonable effort and that Isabelle/HOL in general and its Measure Theory library in particular are suitable for it.

Appendix

A Notation and Auxiliary Functions

A.1 General notation

Notation	Name / Description	Definition
$f\ x$	function application	$f(x)$
$f\ `\ X$	image set	$f(X)$ or $\{f(x) \mid x \in X\}$
$\lambda x.\ e$	lambda abstraction	$x \mapsto e$
undefined	arbitrary value	
Suc	successor of a natural number	$+1$
case_nat $x\ f\ y$	case distinction on natural number	$\begin{cases} x & \text{if } y = 0 \\ f(y-1) & \text{otherwise} \end{cases}$
$[]$	Nil	empty list
$x \# xs$	Cons	prepend element to list
$xs\ @\ ys$	append lists	
map $f\ xs$	applies f to all list elements	$[f(x) \mid x \leftarrow xs]$
merge $V\ V\ (\rho, \sigma)$	merging disjoint states	$\begin{cases} \rho\ x & \text{if } x \in V \\ \sigma\ y & \text{if } x \in V' \\ \text{undefined} & \text{otherwise} \end{cases}$
$y \bullet f$	add de Bruijn variable to scope	see section 2.1.2
$\langle P \rangle$	indicator function	1 if P is true, 0 otherwise
$\int x.\ f\ x\, \partial\mu$	Lebesgue integral	$\int f(x)\, \mathrm{d}\mu(x)$
$\int^{+} x.\ f\ x\, \partial\mu$	Lebesgue integral on nonnegative part	$\int \max(0, f(x))\, \mathrm{d}\mu(x)$
Meas	category of measurable spaces	see section 2.2.2
$\mathbb{S}$	kernel space functor	see section 2.2.3
return	monadic return (η) in the Giry monad	see section 2.2.4
join	monadic join (μ) in the Giry monad	see section 2.2.4
$\gg=$	monadic bind in the Giry monad	see section 2.2.4
do $\{\ldots\}$	monadic "do" syntax	see section 2.2.5
density $M\ f$	measure with density	result of applying density f to M
distr $M\ N\ f$	pushforward measure / image measure	$(B,\ \mathcal{B},\ \lambda X.\ \mu(f^{-1}(X)))$ for $M = (A, \mathcal{A}, \mu)$, $N = (B, \mathcal{B}, \mu')$

A.2 Semantics notation and general auxiliary functions

Notation/Function	Description
$\Gamma \vdash e : t$	source language typing, see figure 4
$\Gamma \vdash_c e : t$	target language typing, see figure 8
$\Upsilon \vdash_d e \Rightarrow f$	abstract compiler, see section 4.2
$\Upsilon \vdash_c e \Rightarrow f$	concrete compiler, see section 5.2
$e : t \Rightarrow_c f$	"compiles to" predicate, see section 5.4
op_sem	operator semantics, see figure 1
dist_param_type *dst*	parameter type of the builtin distribution *dst*
dist_result_type *dst*	result type of the builtin distribution *dst*
dist_measure *dst* x	builtin distribution *dst* with parameter x
dist_dens *dst* x y	density of the builtin distribution *dst* w. parameter x at value y
expr_sem	source language semantics, see figure 6
expr_sem_rf	semantics for randomfree expression, see section 3.4
cexpr_sem	target language semantics, see figure 9
type_of Γ e	the unique t such that $\Gamma \vdash e : t$
val_type v	the type of value v, e. g. val_type $(\text{IntVal } 42) = \text{INTEG}$
type_universe t	the set of values of type t
countable_type t	true iff type_universe t is a countable set
free_vars e	the free (i. e. non-bound) variables in the expression e
randomfree e	true iff e does not contain Random or Fail
extract_real x	returns y for $x = \text{RealVal } y$ (analogous for int, pair, etc.)
embed_measure M f	embedding measure M with inj. function f, see section 3.2.1
stock_measure t	stock measure for type t, see section 3.2.2
state_measure Γ V	measure on states, see section 3.2.2
return_val v	return (stock_measure (val_type v)) v
null_measure M	measure with same measurable space as M, but 0 for all sets
branch_prob	probability of being in the current branch, see section 4.2
marg_dens	marginal density of a random variable, see section 4.2
marg_dens2	marginal density of two random variables, see section 4.2
has_subprob_density M N f	f is a valid density function, applying density f to N yields M, and M is a subprobability measure. See section 4.2
has_parametrized_subprob_density R M N f	As before, but with additional parameter from R. See section 4.2

A.3 Target language auxiliary functions

For the definitions of the following functions, see the corresponding Isabelle theory file `PDF_Target_Semantics.thy`. We will not print them here since they are rather technical.

In this section, x will always be a variable, v will always be a value, and e and e' will always be target language expressions. f and g will always be target language expressions interpreted as functions, i. e. with an implicit λ abstraction around them.

All functions in the table take de Bruijn indices into account, i. e. they will shift variables when an they enter the scope of an integral.

Function	Description
map_vars h e	transforms all variables in e with the function h
cexpr_subst x e e'	substitutes e for x in e'
cexpr_subst_val e v	substitutes v for CVar 0 in e, effectively applying e interpreted as a function to the argument v.
cexpr_comp f g	function composition, i. e. $\lambda x.\ f\ (g\ x)$
$f \circ_c g$	alias for $cexpr\_comp$
expr_rf_to_cexpr e	converts a randomfree source language expression into a target language expression, see section 3.5
integrate_var Γ x e	integrates e over the free variable x
integrate_vars Γ xs e	integrates e over the free variables in the list xs
branch_prob_cexpr $(vs, vs', \Gamma, \delta)$ marg_dens_cexpr Γ vs x δ marg_dens2_cexpr Γ vs x y δ	returns an expression that computes the branch_prob/ marg_dens/marg_dens2
dist_dens_cexpr dst e e'	returns an expression that computes the density of the builtin distribution dst, parametrised with e and evaluated at e'

B References

[AHS14] Jeremy Avigad, Johannes Hölzl, and Luke Serafin. A formally verified proof of the Central Limit Theorem. *CoRR*, abs/1405.7012, 2014.

[APM06] Philippe Audebaud and Christine Paulin-Mohring. Proofs of randomized algorithms in Coq. In *Mathematics of Program Construction*, volume 4014 of *Lecture Notes in Computer Science*, pages 49–68. Springer Berlin Heidelberg, 2006.

[BAVG12] Sooraj Bhat, Ashish Agarwal, Richard Vuduc, and Alexander Gray. A type theory for probability density functions. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 545–556, New York, NY, USA, 2012. ACM.

[BBGR13] Sooraj Bhat, Johannes Borgström, Andrew D. Gordon, and Claudio Russo. Deriving probability density functions from probabilistic functional programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 7795 of *Lecture Notes in Computer Science*, pages 508–522. Springer Berlin Heidelberg, 2013.

[BBGR14] Sooraj Bhat, Johannes Borgström, Andrew D. Gordon, and Claudio Russo. Deriving probability density functions from probabilistic functional programs. In *TODO*. 2014.

[Coc12] David Cock. Verifying probabilistic correctness in Isabelle with pGCL. In *Proceedings of the 7th Systems Software Verification*, pages 1–10, November 2012.

[Coc14] David Cock. pGCL for Isabelle. *Archive of Formal Proofs*, July 2014. `http://afp.sf.net/entries/pGCL.shtml`, Formal proof development.

[Dob07] Ernst-Erich Doberkat. *Stochastic Relations: Foundations for Markov Transition Systems*. Studies in Informatics. Chapman & Hall/CRC, 2007.

[Dob08] Ernst-Erich Doberkat. Basing Markov transition systems on the Giry monad. `http://www.informatics.sussex.ac.uk/events/domains9/Slides/Doberkat_GiryMonad.pdf`, 2008.

[Gir82] Michèle Giry. A categorical approach to probability theory. In *Categorical Aspects of Topology and Analysis*, volume 915 of *Lecture Notes in Mathematics*, pages 68–85. Springer Berlin Heidelberg, 1982.

[HMM05] Joe Hurd, Annabelle McIver, and Carroll Morgan. Probabilistic guarded commands mechanized in HOL. *Electron. Notes Theor. Comput. Sci.*, 112:95–111, January 2005.

[Hö12] Johannes Hölzl. Construction and stochastic applications of measure spaces in higher-order logic. PhD thesis, Technische Universität München, Institut für Informatik, 2012.