A Verified Compiler for Probability Density Functions

Manuel Eberl, Johannes Hölzl, and Tobias Nipkow

Fakultät für Informatik, Technische Universität München, Germany

Abstract. Bhat *et al.* developed an inductive compiler that computes density functions for probability spaces described by programs in a probabilistic functional language. 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. Together with Isabelle's code generation for inductive predicates, this yields a fully verified, executable density compiler. The proof is done in two steps: First, an abstract compiler working with abstract functions modelled directly in the theorem prover's logic is defined and proved sound. Then, this compiler is refined to a concrete version that returns a target-language expression.

1 Introduction

Random distributions of practical significance can often be expressed as probabilistic functional programs. When studying a random distribution, 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 *Markov-chain Monte Carlo* (MCMC).

Bhat *et al.* [5] presented a compiler that computes the probability density function of a program in the probabilistic functional language Fun. Fun is a small functional language with basic arithmetic, Boolean logic, product and sum types, conditionals, and a number of built-in discrete and continuous distributions. It does not support lists or recursion. 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. A correctness proof for the compiler is sketched.

Bhat *et al.* [4] stated that their eventual goal is the formal verification of this compiler in a theorem prover. We have verified such a compiler for a similar probabilistic functional language in the interactive theorem prover Isabelle/HOL [18, 19]. Our contributions are the following:

– a formalisation of the source language, target language (whose semantics had previously not been given precisely), and the compiler on top of a foundational theory of measure spaces
– a formal verification of the correctness of the compiler
– executable code for the compiler using Isabelle's code generator

In the process, we uncovered an incorrect generalisation of one of the compiler rules in the draft of an extended version of the paper by Bhat *et al.* [6].

The complete formalisation is available online [13].

In this paper, we focus entirely on the correctness proof; for more motivation and applications, the reader should consult Bhat *et al.* [5].

1.1 Related work

Park *et al.* [20] developed a probabilistic extension of Objective CAML called $\lambda_{\bigcirc}$. While Bhat *et al.* generate density functions of functional programs, Park *et al.* generate *sampling functions*. This approach allows them to handle much more general distributions, even recursively-defined ones and distributions that do not have a density function, but it does not allow precise reasoning about these distributions (such as determining the exact expectation). No attempt at formal verification is made.

Several formalisations of probabilistic programs already exist. Hurd [16] formalises programs as random variables on infinite streams of random bits. Hurd *et al.* [17] and Cock [8, 9] both formalise pGCL, an imperative programming language with probabilistic and non-deterministic choice. Audebaud and Paulin-Mohring [1] verify probabilistic functional programs in Coq [3] using a shallow embedding based on the Giry monad on discrete probability distributions. All these program semantics support only discrete distributions – even the framework by Hurd [16], although it is based on measure theory.

Our work relies heavily on a formalisation of measure theory by Hölzl [15] and some material developed for the proof of the Central Limit Theorem by Avigad *et al.* [2].

1.2 Outline

Section 2 explains the notation and gives a brief overview of the mathematical basis. Section 3 defines the source and target language. Section 4 defines the abstract compiler and gives a high-level outline of the soundness proof. Section 5 explains the refinement of the abstract compiler to the concrete compiler and the final correctness result and evaluates the compiler on a simple example.

2 Preliminaries

2.1 Typographical notes

We will use the following typographical conventions in mathematical formulæ:

- Constants, functions, datatype constructors, and types will be set in slanted font.
- Free and bound variables (including type variables) are set in italics.
- Isabelle keywords are set in bold font: **lemma**, **datatype**, etc.
- σ-algebras are set in calligraphic font: $\mathcal{A}$, $\mathcal{B}$, $\mathcal{M}$, etc.
- File names of Isabelle theories are set in a monospaced font: `PDF_Compiler.thy`.

2.2 Isabelle/HOL basics

The formalisations presented in this paper employ the Isabelle/HOL theorem prover. We will aim to stay as close to the Isabelle formalisation syntactically as possible. In this section, we give an overview of the syntactic conventions we use to achieve this.

The term syntax follows the λ-calculus, i.e. function application is juxtaposition as in $f\ t$. The notation $t :: \tau$ means that the term t has type τ. Types are built from the base types *bool*, *nat* (natural numbers), *real* (reals), *ereal* (extended reals, i.e. $\mathit{real} \cup \{+\infty, -\infty\}$), and type variables ($\alpha$, β, etc) via the function type constructor $\alpha \to \beta$ or the set type constructor $\alpha\ \mathit{set}$. The constant $\mathit{undefined} :: \alpha$ describes an arbitrary element for each type α. There are no further axioms about it, expecially no defining equation. $f\ `\ X$ is the image set of X under f: $\{f\ x \mid x \in X\}$. We write $\langle P \rangle$ for the indicator of P: 1 if P is true, 0 otherwise.

Because we represent variables by de Bruijn indices [7], variable names are natural numbers and program states are functions of type $\mathit{nat} \to \alpha$. As HOL functions are total, we use *undefined* to fill in the unused places, e.g. $(\lambda x.\ \mathit{undefined})$ describes the empty state. Prepending an element x to a state $\omega :: \mathit{nat} \to \alpha$ is written as $x \bullet \omega$, i.e. $(x \bullet \omega)\ 0 = x$ and $(x \bullet \omega)\ (n+1) = \omega\ n$. The function *merge* merges two states with given domains:

$$\mathit{merge}\ V\ V'\ (\rho, \sigma) = \begin{cases} \rho\ x & \text{if } x \in V \\ \sigma\ y & \text{if } x \in V' \setminus V \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

Notation. We use Γ to denote a type environment, i. e. a function from variable names to types, and σ to denote a state.

The notation $t \bullet \Gamma$ (resp. $v \bullet \sigma$) denotes the insertion of a new variable with the type t (resp. value v) into a typing environment (resp. state). We use the same notation for inserting a new variable into a set of variables, shifting all other variables, i. e.: $0 \bullet V = \{0\} \cup \{y + 1 \mid y \in V\}$

2.3 Measure theory in Isabelle/HOL

We use Isabelle's measure theory, as described in [15]. This section gives an introduction of the measure-theoretical concepts used in this paper. The type $\alpha\ \mathit{measure}$ describes a measure over the type α. Each measure μ is described by the following three projections: $\mathit{space}\ \mu :: \alpha\ \mathit{set}$ is the space, $\mathit{sets}\ \mu :: \alpha\ \mathit{set}\ \mathit{set}$ are the measurable sets, and $\mathit{measure}\ \mu :: \alpha\ \mathit{set} \to \mathit{ereal}$ is the measure function valid on the measurable sets. The type $\alpha\ \mathit{measure}$ guarantees that the measurable sets are a σ-algebra and that $\mathit{measure}\ \mu$ is a non-negative and σ-additive function. In the following we will always assume that the occuring sets and functions are measurable. We also provide integration over measures:

$\int :: (\alpha \to \mathit{ereal}) \to \alpha\ \mathit{measure} \to \mathit{ereal}$, we write $\int x.\, f\, x\, \partial\mu$. This is the *non-negative Lebesgue integral*; for this integral, most rules do not require integrable functions – measurability is enough.

We write $(A, \mathcal{A}, \mu)$ for a measure with space A, measurable sets $\mathcal{A}$ and the measure function $\mu :: \alpha\ \mathit{set} \to \mathit{ereal}$. If we are only intersted in the measurable space we write $(A, \mathcal{A})$. When constructing a measure in this way, the measure function is the constant zero function.

Sub-probability spaces. A sub-probability space is a measure $(A, \mathcal{A}, \mu)$ with $\mu\ A \le 1$. 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.

Constructing measures. The semantics of our language will be given in terms of measures. We have the following functions to construct measures:

Counting:	$\mathit{measure}\ (\mathit{count}\ A)\ X = \lvert X \rvert$
Lebesgue-Borel:	$\mathit{measure}\ \mathit{borel}\ [a; b] = b - a$
With density:	$\mathit{measure}\ (\mathit{density}\ \mu\ f)\ X = \int x.\ f\ x \cdot \langle x \in X \rangle \partial\mu$
Push-forward:	$\mathit{measure}\ (\mathit{distr}\ \mu\ \nu\ f)\ X = \mathit{measure}\ \mu\ \{x \mid f\ x \in X\}$
Product:	$\mathit{measure}\ (\mu \otimes \nu)\ (A \times B) = \mathit{measure}\ \mu\ A \cdot \mathit{measure}\ \nu\ B$
Indexed product:	$\mathit{measure}\ (\bigotimes_{i \in I} \mu_i)\ (\times_{i \in I} A_i) = \prod_{i \in I} \mathit{measure}\ \mu_i\ A_i$
Embedding:	$\mathit{measure}\ (\mathit{embed}\ \mu\ f)\ X = \mathit{measure}\ \mu\ \{x \mid f\ x \in X\}$

The push-forward measure and the embedding of a measure have different measurable sets. The σ-algebra of the push-forward measure *distr* $\mu\ \nu\ f$ is given by ν. The measure is only well-defined when f is μ-ν-measurable. The σ-algebra of *embed* $\mu\ f$ is generated by the sets $f[A]$ for A μ-measurable. The embedding measure is well-defined when f is injective.

2.4 Giry monad

The category theory part of this section is based mainly on a presentation by Ernst-Erich Doberkat [11]. For a more detailed introduction, see his textbook [10] or the original paper by Michèle Giry [14]. Essentially the Giry monad is a monad on measures.

The category $\mathfrak{Meas}$ has measurable spaces as objects and measurable maps as morphisms. This category forms the basis on which we will define the Giry monad. In Isabelle/HOL, the objects are represented by the type *measure*, and the morphism are represented as regular functions. When we mention a measurable function, we explicitly need to mention the measurable spaces representing the domain and the range.

The *sub-probability* functor $\mathbb{S}$ is an endofunctor on $\mathfrak{Meas}$. It maps a measurable space $\mathcal{A}$ to the measurable space of all sub-probabilities on $\mathcal{A}$. Given a measurable space $(A, \mathcal{A})$, we consider the set of all sub-probability measures on $(A, \mathcal{A})$:

$$M = \{\mu \mid \mu \text{ is a sub-probability measure on } (A, \mathcal{A})\}$$

The measurable space $\mathbb{S}(A, \mathcal{A})$ is the smallest measurable space on M that fulfils the following property:

$$\text{For all } X \in \mathcal{A},\ \ (\lambda\mu.\ \mathit{measure}\ \mu\ X)\ \text{ is }\ \mathbb{S}(A, \mathcal{A})\text{-Borel-measurable}$$

A $\mathcal{M}$-$\mathcal{N}$-measurable function f is mapped with $\mathbb{S}(f) = \lambda\mu.\ \mathit{distr}\ \mu\ \mathcal{N}\ f$, where all μ are sub-probability measures on $\mathcal{M}$.

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* 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:

$$\begin{aligned}
&\mathit{return} :: \alpha \to \alpha\ \mathit{measure}\\
&\mathit{return}\ x := \lambda X. \begin{cases} 1 & \text{if } x \in X \\ 0 & \text{otherwise} \end{cases}
\end{aligned}$$

$$\begin{aligned}
&\mathit{bind} :: \alpha\ \mathit{measure} \to (\alpha \to \beta\ \mathit{measure}) \to \beta\ \mathit{measure}\\
&\mu \gg\!= f := \lambda X. \int x.\ f(x)(X)\, \partial\mu
\end{aligned}$$

The actual definitions of *return* and *bind* in Isabelle are slightly more complicated due to Isabelle's simple type system. In informal mathematics, a function typically has attached to it the information of what its domain and codomain are and what the corresponding measurable spaces are; with simple types, this is not directly possible and requires some tricks in order to infer the carrier set and the σ-algebra of the result.

The "do" syntax. For better readability, we employ a Haskell-style "**do** notation" for operations in the Giry monad. The syntax of this notation is defined recursively, where M stands for a monadic expression and $\langle \mathit{text} \rangle$ stands for arbitrary "raw" text:

$$\mathbf{do}\ \{M\}\ \hat{=}\ M \qquad \mathbf{do}\ \{x \leftarrow M;\ \langle \mathit{text} \rangle\}\ \hat{=}\ M \gg\!= (\lambda x.\ \mathbf{do}\ \{\langle \mathit{text} \rangle\})$$

3 Source and Target Language

The source language used in the formalisation was modelled after the language Fun described by Bhat *et al.* [5]; 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.
– No sum types are supported. Consequently, the **match-with** construct is replaced with an *IF-THEN-ELSE*. 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.

In the following subsections, 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 1 shows the types and values that exist in our languages.[1] Additionally, standard arithmetical and logical operators exist.

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 non-standard definitions for non-total operations such as division, the logarithm, and the square root. Non-totality could also be handled by implementing operators in the Giry monad by 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. This, however, would probably complicate many proofs significantly.

To increase readability, we will use the following abbreviations:

– *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.

datatype *pdf_type* =
  UNIT | $\mathbb{B}$ | $\mathbb{Z}$ | $\mathbb{R}$ | *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*

Fig. 1: Types and values in source and target language

[1] Note that *bool*, *int*, and *real* stand for the respective Isabelle types, whereas $\mathbb{B}$, $\mathbb{Z}$, and $\mathbb{R}$ stand for the source-/target-language types.

Table 1: Auxiliary functions

Function	Description
op_sem op v	semantics of operator *op* applied to *v*
op_type op t	result type of operator *op* for input type *t*
dist_param_type dst	parameter type of the built-in distribution *dst*
dist_result_type dst	result type of the built-in distribution *dst*
dist_measure dst x	built-in distribution *dst* with parameter *x*
dist_dens dst x y	density of the built-in distribution *dst* w. parameter *x* at value *y*
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* (*IntVal* 42) = *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 variables in the expression *e*
e det	*True* iff *e* does not contain *Random* or *Fail*
extract_real x	returns *y* for *x* = *RealVal y* (analogous for int, pair, etc.)
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

3.2 Auxiliary definitions

A number of auxiliary definitions are used in the definition of the semantics; Table 1 lists some simple auxiliary functions. Additionally, the following two notions require a detailed explanation:

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*, $\mathbb{B}$, $\mathbb{Z}$: the count measure over the corresponding type universes
– For type $\mathbb{R}$: the embedding of the Lebesgue-Borel measure on $\mathbb{R}$ with *RealVal*
– For $t_1 \times t_2$: the embedding of the product measure

$$\mathit{stock\_measure}\ t_1 \otimes \mathit{stock\_measure}\ t_2$$

with $\lambda(v_1, v_2).\ <\!|v_1, v_2|\!>$

Note that in order to save space and increase readability, we will often write $\int x.\ f\ x\,\partial t$ instead of $\int x.\ f\ x\,\partial \mathit{stock\_measure}\ t$ in integrals.

State measure. Using the stock measure, we construct a measure on states in the context of a typing environment Γ. A state σ is *well-formed* w. r. t. to V and Γ if it maps every variable $x \in V$ to a value of type $\Gamma\ x$ and every variable $\notin V$ to *undefined*. We 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 the finite product measure of the stock measures of the types of the variables:

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

3.3 Source language

datatype *expr* =
  Var nat | *Val val* | *LET expr IN expr* | *pdf_operator* \$ *expr* | *<expr, expr>* |
  Random pdf_dist | *IF expr THEN expr ELSE expr* | *Fail pdf_type*

Fig. 2: Source language expressions

Figure 2 shows the syntax of the source language. It contains variables (de Bruijn), values, *LET*-expressions (again de Bruijn), operator application, pairs, sampling a parametrised built-in random distribution, *IF-THEN-ELSE* and failure. We omit the constructor *Val* when its presence is obvious from the context.

Figures 3 and 4 show the typing rules and the monadic semantics of the source language.

Figure 5 shows the built-in distributions of the source language, their parameter types and domains, the types of the random variables they describe, and their density functions in terms of their parameter. When given a parameter *outside* their domain, they return the null measure. We support the same distributions as Bhat *et al.*, except for the Beta and Gamma distributions (merely because we have not formalised them yet).

$$\frac{}{\Gamma \vdash \mathit{Val}\ v : \mathit{val\_type}\ v} \qquad \frac{}{\Gamma \vdash \mathit{Var}\ x : \Gamma\ x} \qquad \frac{}{\Gamma \vdash \mathit{Fail}\ t : t}$$

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

$$\frac{\Gamma \vdash b : \mathbb{B} \qquad \Gamma \vdash e_1 : t \qquad \Gamma \vdash e_2 : t}{\Gamma \vdash \mathit{IF}\ b\ \mathit{THEN}\ e_1\ \mathit{ELSE}\ e_2 : t} \qquad \frac{\Gamma \vdash e_1 : t_1 \qquad t_1 \bullet \Gamma \vdash e_2 : t_2}{\Gamma \vdash \mathit{LET}\ e_1\ \mathit{IN}\ e_2 : t_2}$$

$$\frac{\Gamma \vdash e : \mathit{dist\_param\_type}\ dst}{\Gamma \vdash \mathit{Random}\ dst\ e : \mathit{dist\_result\_type}\ dst}$$

Fig. 3: Typing rules of the source language

```
$expr\_sem :: state \to expr \to val\ measure$
$expr\_sem\ \sigma\ (Val\ v) = return\_val\ v$
$expr\_sem\ \sigma\ (Var\ x) = return\_val\ (\sigma\ x)$
$expr\_sem\ \sigma\ (LET\ e_1\ IN\ e_2) =$
    $\mathbf{do}\ \{v \leftarrow expr\_sem\ \sigma\ e_1;\ expr\_sem\ (v \bullet \sigma)\ e_2\}$
$expr\_sem\ \sigma\ (op\ \$\ e) =$
    $\mathbf{do}\ \{v \leftarrow expr\_sem\ \sigma\ e;\ return\_val\ (op\_sem\ op\ v)\}$
$expr\_sem\ \sigma\ <e_1, e_2> =$
    $\mathbf{do}\ \{v_1 \leftarrow expr\_sem\ \sigma\ e_1;\ v_2 \leftarrow expr\_sem\ \sigma\ e_2;$
        $return\_val\ <|v_1, v_2|>\}$
$expr\_sem\ \sigma\ (IF\ b\ THEN\ e_1\ ELSE\ e_2) =$
    $\mathbf{do}\ \{b' \leftarrow expr\_sem\ \sigma\ b;$
        $expr\_sem\ \sigma\ (\mathbf{if}\ b' = TRUE\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2)\}$
$expr\_sem\ \sigma\ (Random\ dst\ e) =$
    $\mathbf{do}\ \{p \leftarrow expr\_sem\ \sigma\ e;\ dist\_measure\ dst\ p\}$
$expr\_sem\ \sigma\ (Fail\ t) = null\_measure\ (stock\_measure\ t)$
```

Fig. 4: Semantics of the source language

Distribution	Param.	Domain	Type	Density function
Bernoulli	$\mathbb{R}$	$p \in [0;1]$	$\mathbb{B}$	$\lambda x.\ \begin{cases} p & \text{for } x = \mathit{TRUE} \\ 1-p & \text{for } x = \mathit{FALSE} \end{cases}$
UniformInt	$\mathbb{Z} \times \mathbb{Z}$	$p_1 \leq p_2$	$\mathbb{Z}$	$\lambda x.\ \dfrac{\langle x \in [p_1; p_2] \rangle}{p_2 - p_1 + 1}$
UniformReal	$\mathbb{R} \times \mathbb{R}$	$p_1 < p_2$	$\mathbb{R}$	$\lambda x.\ \dfrac{\langle x \in [p_1; p_2] \rangle}{p_2 - p_1}$
Gaussian	$\mathbb{R} \times \mathbb{R}$	$p_2 > 0$	$\mathbb{R}$	$\lambda x.\ \dfrac{1}{\sqrt{2\pi p_2^2}} \cdot \exp\left(-\dfrac{(x-p_1)^2}{2p_2^2}\right)$
Poisson	$\mathbb{R}$	$p \geq 0$	$\mathbb{Z}$	$\lambda x.\ \begin{cases} \exp(-p) \cdot p^x / x! & \text{for } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$

Fig. 5: Built-in 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.

3.4 Deterministic expressions

We call an expression e *deterministic* (written as "e *det*") if it contains no occurrence of *Random* or *Fail*. Such expressions are of particular interest: if all their free variables have a fixed value, they return precisely one value, so we can define a function *expr_sem_rf* [2] that, when given a state σ and a deterministic expression e, returns this single value. This function can be seen as a non-monadic analogue to *expr_sem* and its definition is therefore analogous and is not shown here. The function *expr_sem* has the following property (assuming that e is deterministic and well-typed and σ is a valid state):

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

This property will enable us to convert deterministic source-language expressions into "equivalent" target-language expressions.

3.5 Target language

The target language is again modelled very closely after the one by Bhat *et al.* [5]. The type system and the operators are the same as in the source language. The

[2] In Isabelle, the expression *randomfree* is used instead of *deterministic*, hence the "rf" suffix. This is in order to emphasise the syntactical nature of the property. Note that a syntactically deterministic expression is not truly deterministic if the variables it contains are randomised over, which can be the case.

key difference is that the *Random* construct has been replaced by an integral. As a result, while expressions in the source language return a measure space, expressions in the target language always return a single value.

Since our source language lacks sum types, so does our target language. Additionally, our target language differs from the one 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, on the other hand, 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 6, 7, and 8 show the syntax, typing rules, and semantics of the target language.

The matter of target-language semantics in the papers by Bhat *et al.* is somewhat unclear. In the 2012 POPL paper [4], the only semantics given for the target language is a vague denotational rule for the integral. In the 2013 TACAS paper [5], no target-language semantics is given at all; it is only said that "standard CBV small-step evaluation" is used. The extended version of this paper currently submitted for publication [6] indeed gives some small-step evaluation rules, but only for simple cases. In particular, none of these publications give the precise rules for evaluating integral expressions. It is quite unclear to us how small-step evaluation of integral expressions is possible in the first place. Another issue is how to handle evaluation of integral expressions where the integrand evaluates to $\perp$ for some values.[3]

Converting deterministic expressions. The auxiliary function *expr_rf_to_cexpr*, which will be used in some rules of the compiler that handle deterministic expressions, is of particular interest. We mentioned earlier that deterministic source-language expressions can be converted to equivalent target-language expressions.[4] This function does precisely that. Its definition is mostly obvious,

[3] We do not have this problem since in our target language, as mentioned before, evaluation cannot fail.

[4] Bhat *et al.* say that a deterministic expression "is also an expression in the target language syntax, and we silently treat it as such" [5]

datatype *cexpr* =

$\quad$ *CVar nat* | *CVal val* | *pdf_operator* $\$_{\mathrm{c}}$ *cexpr* | $<$*cexpr*, *cexpr*$>_{\mathrm{c}}$ |

$\quad$ IF_{c} *cexpr THEN cexpr ELSE cexpr* | $\int_{\mathrm{c}} \mathit{cexpr}\, \partial \mathit{pdf\_type}$

Fig. 6: Target language expressions

$$\frac{}{\Gamma \vdash_{\mathrm{c}} \mathit{CVal}\ v\ :\ \mathit{val\_type}\ v} \qquad \frac{}{\Gamma \vdash_{\mathrm{c}} \mathit{CVar}\ x\ :\ \Gamma\ x}$$

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

$$\frac{\Gamma \vdash_{\mathrm{c}} b\ :\ \mathbb{B} \qquad \Gamma \vdash_{\mathrm{c}} e_1\ :\ t \qquad \Gamma \vdash_{\mathrm{c}} e_2\ :\ t}{\Gamma \vdash_{\mathrm{c}} \mathit{IF}_{\mathrm{c}}\ b\ \mathit{THEN}\ e_1\ \mathit{ELSE}\ e_2\ :\ t} \qquad \frac{t \bullet \Gamma \vdash_{\mathrm{c}} e\ :\ \mathbb{R}}{\Gamma \vdash_{\mathrm{c}} \int_{\mathrm{c}} e\, \partial t\ :\ \mathbb{R}}$$

Fig. 7: Typing rules for target language

cexpr_sem :: *state* → *cexpr* → *val*

cexpr_sem σ (*CVal* v) = v

cexpr_sem σ (*CVar* x) = $\sigma\ x$

cexpr_sem σ $<e_1, e_2>_{\mathrm{c}}$ = $<|$*cexpr_sem* σ e_1, *cexpr_sem* σ $e_2|>$

cexpr_sem σ *op* $\$_{\mathrm{c}}$ e = *op_sem op* (*cexpr_sem* σ e)

cexpr_sem σ (IF_{c} b *THEN* e_1 *ELSE* e_2) =

$\quad$ (**if** *cexpr_sem* σ b = *TRUE* **then** *cexpr_sem* σ e_1 **else** *cexpr_sem* σ e_2)

cexpr_sem σ ($\int_{\mathrm{c}} e\, \partial t$) =

$\quad$ *RealVal* ($\int x.$ *extract_real* (*cexpr_sem* ($x \bullet \sigma$) e) ∂ *stock_measure* t)

Fig. 8: Semantics of target language

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.

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

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

4 Abstract compiler

The correctness proof is done in two steps using a refinement approach: first, we define and prove correct an *abstract compiler* that returns the density function as an abstract mathematical function. We then define an analogous *concrete compiler* that returns a target-language expression and show that it is a *refinement* of the abstract compiler, which will allow us to lift the correctness result from the latter to the former.

4.1 Density contexts

First, we define the notion of a *density context*, which holds the 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* $(V \cup V')\ \Gamma$) 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* $V'\ \Gamma$) be a parameter state. We write

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

for the measure obtained by taking *state_measure* $V\ \Gamma$, 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:

$$\begin{aligned}&\mathit{dens\_ctxt\_measure}\ ::\ \mathit{dens\_ctxt} \rightarrow \mathit{state} \rightarrow \mathit{state\ measure}\\ &\mathit{dens\_ctxt\_measure}\ (V, V', \Gamma, \delta)\ \rho\ =\ \mathit{density}\ (\mathit{distr}\ (\mathit{state\_measure}\ V\ \Gamma)\\ &\qquad\qquad (\mathit{state\_measure}\ (V \cup V')\ \Gamma)\ (\lambda\sigma.\ \mathit{merge}\ V\ V'\ (\sigma, \rho)))\ \delta\end{aligned}$$

Informally, *dens_ctxt_measure* describes the measure obtained by integrating over the variables $\{v_1, \ldots, v_m\} = V$ while treating the variables $\{v'_1, \ldots, v'_n\} =$

V' as parameters. The evaluation of an expression e with variables from $V \cup V'$ in this context is effectively a function

$$\lambda v'_1 \ldots v'_n.\ \int v_1.\ \ldots \int v_m.\ \mathit{expr\_sem}\ (v_1, \ldots, v_m, v'_1, \ldots, v'_n)\ e\ \cdot \\ \delta\ (v_1, \ldots, v_m, v'_1, \ldots, v'_n)\ \partial\mathit{\Gamma}\ v_1 \ldots \partial\mathit{\Gamma}\ v_m\ .$$

A density context is *well-formed* (predicate *density_context* in Isabelle) if:

- V and V' are finite and disjoint
- $\delta\ \sigma \geq 0$ for any $\sigma \in \mathit{space}\ (\mathit{state\_measure}\ (V \cup V')\ \mathit{\Gamma})$
- δ is Borel-measurable w. r. t. $\mathit{state\_measure}\ (V \cup V')\ \mathit{\Gamma}$
- the measure $\mathit{dens\_ctxt\_measure}\ (V, V', \mathit{\Gamma}, \delta)\ \rho$ is a sub-probability measure for any $\rho \in \mathit{space}\ (\mathit{state\_measure}\ V'\ \mathit{\Gamma})$

4.2 Definition

As a first step, we have implemented an abstract density compiler as an inductive predicate $\mathit{\Upsilon} \vdash_{\mathrm{d}} e \Rightarrow f$, where $\mathit{\Upsilon}$ is a density context, e is a source-language expression and f is a function of type $\mathit{val\ state} \rightarrow \mathit{val} \rightarrow \mathit{ereal}$. 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 very similar to those by Bhat *et al.* [5], except for the following adaptations:

- Bhat *et al.* handle *IF-THEN-ELSE* with the "match" rule for sum types. As 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 LET*-bindings. 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.[5] The functions *marg_dens* and *marg_dens2* compute the marginal density of one and 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.

[5] Additionally, three congruence rules are required for technical reasons. These rules are required because the abstract and the concrete result may differ on a null set and outside their domain.

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

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

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

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

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

HD_RAND
$$\frac{(V, V', \mathit{\Gamma}, \delta) \vdash_{\mathrm{d}} e \Rightarrow f}{\begin{array}{l}(V, V', \mathit{\Gamma}, \delta) \vdash_{\mathrm{d}} \mathit{Random}\ \mathit{dst}\ e \Rightarrow \\ \qquad \lambda\rho\, y.\ \textstyle\int x.\ f\ \rho\ x \cdot \mathit{dist\_dens}\ \mathit{dst}\ x\ y\ \partial\, \mathit{dist\_param\_type}\ \mathit{dst}\end{array}}$$

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

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

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

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

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

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

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

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

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

$$\text{HD\_ADD}\quad \frac{(V, V', \Gamma, \delta) \vdash_{\mathrm{d}} {<}e_1, e_2{>} \Rightarrow f}{(V, V', \Gamma, \delta) \vdash_{\mathrm{d}} e_1 + e_2 \Rightarrow \lambda\rho\, z.\ \int x.\ f\ \rho\ {<}|x, z - x|{>}\ \partial\textit{type\_of}\ \Gamma\ e_1}$$

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

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

Consider the following simple example program:

IF Random *Bernoulli* 0.25 THEN 0 ELSE 1

Applying the abstract compiler yields the following HOL function:

$$\begin{aligned}&\textit{branch\_prob}\ (\emptyset, \emptyset, \Gamma, \lambda\rho.\ \textit{branch\_prob}\ (\emptyset, \emptyset, \Gamma, \lambda\rho.\ 1)\ \rho\ * \\ &\quad \textit{dist\_dens Bernoulli}\ 0.25\ \textit{True})\ \rho \cdot \langle x = 0\rangle\ + \\ &\textit{branch\_prob}\ (\emptyset, \emptyset, \Gamma, \lambda\rho.\ \textit{branch\_prob}\ (\emptyset, \emptyset, \Gamma, \lambda\rho.\ 1)\ \rho\ * \\ &\quad \textit{dist\_dens Bernoulli}\ 0.25\ \textit{False})\ \rho \cdot \langle x = 1\rangle\end{aligned}$$

Since the *branch_prob* in this example is merely the integral over the empty set of variables, this simplifies to:

$$\begin{aligned}&\lambda\rho\ x.\ \textit{dist\_dens Bernoulli}\ 0.25\ \rho\ \textit{True} \cdot \langle x = 0\rangle\ + \\ &\qquad \textit{dist\_dens Bernoulli}\ 0.25\ \rho\ \textit{False} \cdot \langle x = 1\rangle\end{aligned}$$

4.3 Soundness proof

We proved the following soundness result for the abstract compiler: [6]

```
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  $has\_subprob\_density\ (expr\_sem\ \sigma\ e)\ (stock\_measure\ t)\ (f\ (\lambda x.\ undefined))$
```

where *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 sub-probability measure, f is N-Borel-measurable, and f is non-negative on its domain.

The lemma above follows easily from the following generalised lemma:

```
lemma expr_has_density_sound_aux:
assumes  $(V, V', \Gamma, \delta) \vdash_{\mathrm{d}} e \Rightarrow f$   and   $\Gamma \vdash e : t$   and
             $density\_context\ V\ V'\ \Gamma\ \delta$   and   $free\_vars\ e \subseteq V \cup V'$
shows  $has\_parametrized\_subprob\_density\ (state\_measure\ V'\ \Gamma)$
             $(\lambda\rho.\ \mathbf{do}\ \{\sigma \leftarrow dens\_ctxt\_measure\ (V, V', \Gamma, \delta)\ \rho;\ expr\_sem\ \sigma\ e\})$
             $(stock\_measure\ t)\ f$
```

where *has_parametrized_subprob_density* R M N f 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* $(M\ \rho)$ N $(f\ \rho)$ holds.

The proof is by induction on the 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 of the monadic operations 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.* [6] (which we were unaware of while working on our own proof) is analogous to ours, but much more concise due to the fact that side conditions such as measurability, integrability, non-negativity, and so on are not proven explicitly and many important (but uninteresting) steps are skipped or only hinted at.

It should be noted that in the draft of an updated version of their 2013 paper [6], Bhat *et al.* added a scaling rule for real distributions similar to our HD_MULTC rule. However, in the process of our formal proof, we found that their rule was too liberal: while our rule only allows multiplication with a fixed constant, their rule allowed multiplication with any deterministic expression,

[6] 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.

even expressions that may evaluate to 0, but multiplication with 0 always yields the Dirac distribution, which does not have a density function. In this case, the compiler returns a PDF for a distribution that has none, leading to unsoundness. This shows the importance of formal proofs.

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 instead of a HOL function. We use a standard refinement approach to relate the concrete compiler to the abstract one. We thus lift the soundness result on the abstract compiler to an analogous result on the concrete compiler. This shows that the concrete compiler always returns a well-formed target-language expression that represents a density for the sub-probability space described by the source language.

The concrete compilation predicate is written as

$$(vs, vs', \Gamma, \delta) \vdash_{\mathrm{c}} e \Rightarrow f$$

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 – 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. For this reason, we do not show the rules here; they can be found in the Isabelle theory file `PDF_Compiler.thy` [13].

5.3 Refinement

The refinement relates the concrete compilation

$$(vs, vs', \Gamma, \delta) \vdash_{\mathrm{c}} e \Rightarrow f$$

to the abstract compilation

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

In words: we take the abstract compilation predicate and

- the 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 $\Upsilon \vdash_{\mathrm{c}} e \Rightarrow f$, then $\Upsilon' \vdash_{\mathrm{d}} e \Rightarrow f'$, 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 well-formedness of the intermediary expressions needs to be shown, the previously-mentioned congruence lemmas for the abstract compiler need to be applied, and, when integration is involved, non-negativity and integrability have to be shown in order to convert non-negative Lebesgue integrals to general 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:

lemma expr_has_density_cexpr_sound:
assumes $([], [], \Gamma, 1) \vdash_{\mathrm{c}} e \Rightarrow f$ **and** $\Gamma \vdash e : t$ **and** $\mathit{free\_vars}\ e = \emptyset$
shows $\mathit{has\_subprob\_density}\ (\mathit{expr\_sem}\ \sigma\ e)\ (\mathit{stock\_measure}\ t)$
$(\lambda x.\ \mathit{cexpr\_sem}\ (x \bullet \sigma)\ f)$
$\Gamma'\ 0 = t \Longrightarrow \Gamma' \vdash_{\mathrm{c}} f : \mathrm{REAL}$
free_vars $f \subseteq \{0\}$

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

We will now summarise the soundness lemma we have just proven in a more concise manner. For convenience, we define the symbol $e : t \Rightarrow_c f$ (read "e with type t compiles to f"), which includes the well-typedness and groundness requirements on e as well as the compilation result:[7]

$$\begin{aligned} &e : t \Rightarrow_c f \quad \longleftrightarrow \\ &\qquad (\lambda x.\ \mathit{UNIT}) \vdash e\ :\ t\ \wedge\ \mathit{free\_vars}\ e = \emptyset\ \wedge\ ([], [], \lambda x.\ \mathit{UNIT}, 1) \vdash_c e \Rightarrow f \end{aligned}$$

The final soundness theorem for the compiler, stated in Isabelle syntax:[8]

lemma expr_compiles_to_sound:
assumes $e : t \Rightarrow_c f$
shows $\mathit{expr\_sem}\ \sigma\ e = \mathit{density}\ (\mathit{stock\_measure}\ t)\ (\lambda x.\ \mathit{cexpr\_sem}\ (x \bullet \sigma')\ f)$
$\forall x \in \mathit{type\_universe}\ t.\ \mathit{cexpr\_sem}\ (x \bullet \sigma')\ f \geq 0$
$\Gamma \vdash e\ :\ t$
$t \bullet \Gamma' \vdash_c f\ :\ \mathit{REAL}$
$\mathit{free\_vars}\ f \subseteq \{0\}$

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 the measure obtained by evaluating e
- f is non-negative on all input values of type t
- e has no free variables and indeed has type t (in any type context Γ)
- f has no free variable except the parameter (i. e. the variable 0) and is a function from t to REAL[9]

[7] In this definition, the choice of the typing environment is completely arbitrary since the expression contains no free variables.
[8] 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*.
[9] meaning if its parameter variable has type t, it is of type REAL

5.5 Evaluation

Isabelle's code generator allows us to execute our inductively-defined verified compiler using the **values** command[10] or generate code in one of the target languages such as Standard ML or Haskell. As an example on which to test the compiler, we choose the same expression that was chosen by Bhat *et al.* [5]:[11]

$$\begin{gathered}\mathit{LET}\ x = \mathit{Random\ UniformReal}\ {<}0,1{>}\ \mathit{IN}\\ \mathit{LET}\ y = \mathit{Random\ Bernoulli}\ x\ \mathit{IN}\\ \mathit{IF}\ y\ \mathit{THEN}\ x\ +\ 1\ \mathit{ELSE}\ x\end{gathered}$$

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

$$\textbf{values}\ \text{"}\{(t,f)\mid t\ f.\ \ e : t \Rightarrow_{\mathrm{c}} f\}\text{"}$$

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:

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

Further simplification yields the following result:

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

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 β-reduced the compiler output and evaluated constant subexpressions. While such simplification is very useful in practice, we have not automated it yet since it is orthogonal to the focus of our work, the compiler.

[10] Our compiler is inherently non-deterministic 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.

[11] *Val* and *RealVal* were omitted for better readability and symbolic variable names were used instead of de Bruijn indices.

6 Conclusion

6.1 Effort

The formalisation of the compiler took about three months and roughly 10000 lines of Isabelle code (definitions, lemma statements, proofs, and examples) distributed as follows:

Type system and semantics	2900 lines
Abstract compiler	2600 lines
Concrete compiler	1400 lines
General measure theory	3400 lines

As can be seen, a sizeable portion of the work was the formalisation of results from general measure theory, such as integration by substitution and measure embeddings.

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. We proved this, using material from a formalisation of the Central Limit Theorem by Avigad *et al.* [2].

Proving side conditions. Many lemmas from the measure theory library require measurability, integrability, non-negativity, etc. In hand-written proofs, this is often "hand-waved" 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 λ-calculus-like language with symbolic identifiers as a target language. For a paper proof, this is the obvious choice. We chose de Bruijn indices instead; however, this makes handling target language terms less intuitive and requires additional lemmas. Urban's nominal datatypes [21] would have allowed us to work with a more intuitive model, but we would have lost executability of the compiler, which was one of our aims.

6.3 Summary

We formalised the semantics of a 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.* [5]. Then we formally verified 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 (apart from the problem with the scaling rule we discussed earlier), 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. A useful side effect of our work was the formalisation of the Giry Monad, which is useful for formalisations of probabilistic computations in general.

Possible future work includes support for sum types, which should be possible with little effort, and a verified postprocessing stage to automatically simplify the density expression would be desirable.

Acknowledgements. We thank the referees for their helpful comments. We also thank Kristina Magnussen for proofreading the Master's Thesis upon which this work is based [12] and Andrei Popescu for categorical advice. David Cock and Jeremy Avigad helped us with their quick and concise answers to our questions about their work.

References

[1] Audebaud, P., Paulin-Mohring, C.: Proofs of randomized algorithms in Coq. In: Mathematics of Program Construction, Lecture Notes in Computer Science, vol. 4014, pp. 49–68. Springer Berlin Heidelberg (2006), `http://dx.doi.org/10.1007/11783596_6`

[2] Avigad, J., Hölzl, J., Serafin, L.: A formally verified proof of the Central Limit Theorem. CoRR abs/1405.7012 (2014)

[3] Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Springer (2004)

[4] Bhat, S., Agarwal, A., Vuduc, R., Gray, A.: A type theory for probability density functions. In: Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 545–556. POPL '12, ACM, New York, NY, USA (2012), `http://doi.acm.org/10.1145/2103656.2103721`

[5] Bhat, S., Borgström, J., Gordon, A.D., Russo, C.: Deriving probability density functions from probabilistic functional programs. In: Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, vol. 7795, pp. 508–522. Springer Berlin Heidelberg (2013), `http://dx.doi.org/10.1007/978-3-642-36742-7_35`, best Paper Award

[6] Bhat, S., Borgström, J., Gordon, A.D., Russo, C.: Deriving probability density functions from probabilistic functional programs (full version, submitted for publication)

[7] de Bruijn, N.G.: Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. Indagationes Mathematicae 34, 381–392 (1972)

[8] Cock, D.: Verifying probabilistic correctness in Isabelle with pGCL. In: Proceedings of the 7th Systems Software Verification. pp. 1–10 (November 2012)

[9] Cock, D.: pGCL for Isabelle. Archive of Formal Proofs (Jul 2014), http://afp.sf.net/entries/pGCL.shtml, Formal proof development

[10] Doberkat, E.E.: Stochastic relations: foundations for Markov transition systems. Studies in Informatics, Chapman & Hall/CRC (2007)

[11] Doberkat, E.E.: Basing Markov transition systems on the Giry monad. http://www.informatics.sussex.ac.uk/events/domains9/Slides/Doberkat_GiryMonad.pdf (2008)

[12] Eberl, M.: A Verified Compiler for Probability Density Functions. Master's thesis, Technische Universität München (2014), https://in.tum.de/~eberlm/pdfcompiler.pdf

[13] Eberl, M., Hölzl, J., Nipkow, T.: A verified compiler for probability density functions. Archive of Formal Proofs (Oct 2014), http://afp.sf.net/entries/Density_Compiler.shtml, Formal proof development

[14] Giry, M.: A categorical approach to probability theory. In: Categorical Aspects of Topology and Analysis, Lecture Notes in Mathematics, vol. 915, pp. 68–85. Springer Berlin Heidelberg (1982), http://dx.doi.org/10.1007/BFb0092872

[15] Hölzl, J.: Construction and stochastic applications of measure spaces in Higher-Order Logic. PhD thesis, Technische Universität München, Institut für Informatik (2012)

[16] Hurd, J.: Formal Verification of Probabilistic Algorithms. Ph.D. thesis, University of Cambridge (2002)

[17] Hurd, J., McIver, A., Morgan, C.: Probabilistic guarded commands mechanized in HOL. Electron. Notes Theor. Comput. Sci. 112, 95–111 (Jan 2005), http://dx.doi.org/10.1016/j.entcs.2004.01.021

[18] Nipkow, T., Klein, G.: Concrete Semantics with Isabelle/HOL. Springer (forthcoming), http://www.concrete-semantics.org

[19] Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)

[20] Park, S., Pfenning, F., Thrun, S.: A probabilistic language based upon sampling functions. In: Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 171–182. POPL '05, ACM, New York, NY, USA (2005), http://doi.acm.org/10.1145/1040305.1040320

[21] Urban, C.: Nominal techniques in Isabelle/HOL. Journal of Automated Reasoning 40, 327–356 (2008)

Appendix

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
$x \mathbin{\#} xs$	Cons	prepend element to list
$xs \mathbin{@} ys$	list concatenation	
map $f\ xs$	applies f to all list elements	$[f(x) \mid x \leftarrow xs]$
merge $V\ V'\ (\rho, \sigma)$	merging disjoint states	$\begin{cases} \rho\ x & \text{if } x \in V \\ \sigma\ y & \textit{if } x \in V' \\ \textit{undefined} & \text{otherwise} \end{cases}$
$y \bullet f$	add de Bruijn variable to scope	see Sect. 2.2
$\langle P \rangle$	indicator function	1 if P is true, 0 otherwise
$\int x.\ f\ x\ \partial\mu$	Lebesgue integral on non-neg. functions	
$\mathfrak{Meas}$	category of measurable spaces	see Sect. 2.4
$\mathbb{S}$	sub-probability functor	see Sect. 2.4
return	monadic return (η) in the Giry monad	see Sect. 2.4
$\gg=$	monadic bind in the Giry monad	see Sect. 2.4
do $\{\ldots\}$	monadic "do" syntax	see Sect. 2.4
density $M\ f$	measure with density	result of applying density f to M
distr $M\ N\ f$	push-forward/image measure	$(B,\ \mathcal{B},\ \lambda X.\ \mu(f^{-1}(X)))$ for $M = (A, \mathcal{A}, \mu)$, $N = (B, \mathcal{B}, \mu')$