EasyChair Preprint

№ 8385

Program Equivalence in a Typed Probabilistic Call-by-Need Functional Language

David Sabel and Manfred Schmidt-Schauss

July 5, 2022

Program Equivalence in a Typed Probabilistic Call-by-Need Functional Language

David Sabel
LMU Munich
david.sabel@lmu.de

Manfred Schmidt-Schauß
Goethe-University Frankfurt
schauss@ki.informatik.uni-frankfurt.de

We introduce a call-by-need variant of PCF with a binary probabilistic fair choice operator. We define its operational semantics and contextual equivalence as program equivalence, where the expected convergence is only observed on numbers. We investigate another program equivalence that views two closed expressions as distribution-equivalent if evaluation generates the same distribution on the natural numbers. We show that contextual equivalence implies distribution-equivalence. Future work is to provide a full proof of the opposite direction.

1 Introduction

Pure, lazy functional programming languages like Haskell allow a clean mathematical treatment of programs, like equational reasoning. They usually have a large set of correct program transformations. Implementations of those languages use a call-by-need strategy for program execution, i.e. they combine lazy evaluation with sharing to avoid duplicated evaluation of subexpressions (see e.g. [2, 1] for call-by-need lambda-calculi). Probabilistic programming uses programs to express probabilistic and/or stochastic models as a program (see e.g. [4]). Reasoning on such programs means also reasoning on those models. Combining lazy evaluation and functional programming with probabilistic programming thus seems to be a worth-wile investigation. In previous work [16], we introduced an untyped probabilistic call-by-need lambda-calculus and analyzed its contextual semantics. We showed method and techniques to prove correctness of program transformations. In this work, we report on work in progress on a slightly different setting where a typed calculus that includes natural numbers and the *nat*-type is investigated. Our focus is to consider the evaluation of expressions type *nat* and to show properties of the corresponding probability distribution on numbers that is expressed by such a probabilistic program.

Thus we consider a probabilistic variant of PCF [15, 6], a monomorphically typed lambda-calculus with natural numbers, successor- and predecessor-operations, a branching construct (`if-then-else`), a fixpoint-operator to express recursion, a `let`-construct to implement sharing, and binary fair choice $(s \oplus t)$ that randomly chooses between s and t – both with probability 0.5.

For an expression s, summing the probabilities over all successful evaluations results in expected convergence. As program equivalence we use contextual equivalence [14] w.r.t. expected convergence which holds for two expressions s,t if the expected convergence of expression $C[s]$ is the same as the expected convergence of expression $C[t]$ for any program context C.

On type *nat*, the type of natural numbers, one can also observe the expected convergence per value (i.e. per number) which defines a (discrete) probability distribution on all natural numbers. Two expressions of type *nat* are distribution-equivalent if they induce the same probability distribution. Our main goal of this work in progress is to analyse the relation between distribution-equivalence and contextual equivalence. As a first result we show that contextual equivalence implies distribution-equivalence (see Theorem 3.11). We conjecture an implication for the opposite direction, but its proof is ongoing work.

WPTE 2022 informal proceedings

A context lemma restricts the necessity to take into account all contexts for proving contextual equivalences by restricting the regarded contexts to reduction contexts (see [13, 18] for context lemmas in other calculi). We argue that a context lemma holds in the typed setting (see Theorem 3.7). As a direct consequence, some program transformations can be shown to be correct without any effort. In Proposition 3.8 we formulate these correctness results. In this extended abstract, most of the proofs are omitted. They can either be derived in analogous way as in the previous work (for an untyped setting) [16], or they are work-in-progress (we will clearly state, when we can only conjecture a statement).

Related work. Early investigations into probabilistic program calculi are [17, 8]. Operational semantics for a probabilistic lambda-calculus were defined in [11], including call-by-name and call-by-value semantics, but no call-by-need semantics. Contextual equivalence is defined analogously to our definition in [10, 3] for call-by-name and call-by-value calculi. In [9] operational semantics, contextual equivalence, expressive power and termination of a typed call-by-value PCF are discussed. In [19], contextual equivalence for an untyped call-by-value lambda-calculus with continuous random variables is analyzed and program transformations are shown correct. In contrast, the distributions induced by our binary choice operator are discrete. In the categories used by [9], our calculus is a randomized lambda-calculus (it performs random evaluation of the choice operator), while the calculus of [19] is a Bayesian lambda-calculus, since it has sampling and conditioning operators to build continuous probabilistic models.

Outline. In Section 2 the syntax and operational semantics of our language are introduced as well as contextual equivalence, distribution-equivalence in Section 3, the context lemma, correctness of transformations and the relation between contextual and distribution-equivalence are discussed.

2 Syntax and Semantics

2.1 Syntax, Typing, and Operational Semantics

We define the syntax of probabilistic call-by-need PCF, which we call $ProbPCF^{need}$.

Definition 2.1 (Syntax of Expressions and Types)**.** Let *Var* be an infinite, countable set of variables. We use x, y, z, x_i, y_i, z_i for variables of *Var*. The syntax of *expressions* $s, t, r \in Expr$ and *types* $\tau, \rho, \sigma \in Typ$ is given by the following grammar:

$$\begin{array}{lcl} s,t,r \in Expr & ::= & x \mid \lambda x.s \mid (s\ t) \mid (s \oplus t) \mid \texttt{let}\ x = s\ \texttt{in}\ t \mid \texttt{if}\ r\ \texttt{then}\ s\ \texttt{else}\ t \\ & & \mid \texttt{fix}\ s \mid \texttt{pred}\ s \mid \texttt{succ}\ s \mid n \text{ where } n \in \mathbb{N} \\ \tau,\rho,\sigma \in Typ & ::= & nat \mid \tau \to \rho \end{array}$$

The expressions include the lambda-calculus (variables, abstractions, applications), `let`-expressions to share expressions, the binary probabilistic operator $\oplus$, naturals $n = 0, 1, 2\ldots$ (of type *nat*), with operations `pred` and `succ` to compute the predecessor and the successor of a number, a fixpoint-operator `fix`, and branching via `if-then-else`-expressions. Besides basic type *nat*, there are function types $\tau \to \rho$.

Abstractions and `let`-expressions introduce variable binders, where in $\lambda x.s$ the scope of bound variable x is body s, and in $(\texttt{let}\ x = s\ \texttt{in}\ t)$ the scope of the bound variable x is the `in`-expression t. As usual this defines notions of bound and free variables, of bound and free variable occurrences, and a notion of α-renaming (i.e. consistent renaming of binders and their corresponding bound variable occurrences in the scope the binders). Expressions s and t are said to be α-equivalent, if there are α-renamings that make s and t syntactically equal. We use the distinct variable convention and thus assume that the free and bound variables of an expression are pairwise distinct and that all variables on binders are pairwise distinct. The distinct variable convention can always be satisfied by applying α-renamings (with

$$\textsc{Nat}\ \frac{}{n : nat} \quad \textsc{Pred}\ \frac{s : nat}{\texttt{pred}\ s : nat} \quad \textsc{Succ}\ \frac{s : nat}{\texttt{succ}\ s : nat} \quad \textsc{App}\ \frac{s : \tau \to \rho, t : \tau}{(s\ t) : \rho} \quad \textsc{Abs}\ \frac{s : \tau}{\lambda x.s : \Gamma(x) \to \tau}$$

$$\textsc{Fix}\ \frac{s : \rho \to \rho}{(\texttt{fix}\ s) : \rho} \quad \textsc{Var}\ \frac{}{x : \Gamma(x)} \quad \textsc{Prob}\ \frac{s : \rho, t : \rho}{(s \oplus t) : \rho} \quad \textsc{Let}\ \frac{t : \tau, s : \rho, \rho = \Gamma(x)}{(\texttt{let}\ x = s\ \texttt{in}\ t) : \tau} \quad \textsc{If}\ \frac{r : nat, s : \tau, t : \tau}{(\texttt{if}\ r\ \texttt{then}\ s\ \texttt{else}\ t) : \tau}$$

Figure 1: Type check rules

fresh variable name). We also assume that reduction rules or transformations rules implicitly perform α-renaming to obey the distinct variable convention.

For simplicity, we use a type checking system with already fixed (built-in) types of variables, instead of using a type derivation system (with a flexible type environment). So let $\Gamma : \mathit{Var} \to \mathit{Typ}$ be a global typing system that determines the type of every variable.

Definition 2.2 (Typing)**.** A *typing judgment* $s : \tau$ means that s has type τ. The typing rules of $\mathit{ProbPCF}^{need}$ are shown in Fig. 1. We say that a closed expression s is *well-typed* iff $s : \rho$ for some type ρ.

In the following, we use (term) contexts, i.e. expressions that have a hole $[\cdot]$. A general context C has its hole at an arbitrary expressions position, i.e. general contexts are defined by the grammar:

$$\begin{array}{rcl} C & ::= & [\cdot] \mid \lambda x.C \mid (C\ s) \mid (s\ C) \mid (C \oplus s) \mid (s \oplus C) \mid \texttt{let}\ x = s\ \texttt{in}\ C \mid \texttt{let}\ x = C\ \texttt{in}\ s \\ & & \mid \texttt{if}\ C\ \texttt{then}\ s\ \texttt{else}\ t \mid \texttt{if}\ r\ \texttt{then}\ C\ \texttt{else}\ s \mid \texttt{if}\ r\ \texttt{then}\ s\ \texttt{else}\ C \mid \texttt{fix}\ C \mid \texttt{pred}\ C \mid \texttt{succ}\ C \end{array}$$

A *surface context* is a context, where the context hole is not below an abstraction. For a context C and an expression s, we write $C[s]$ for substituting the hole of C with expression s. Concerning typing, a context C can have any type that can be given to the expression $C[x]$, where x is a fresh variable. Since this is non-deterministic we label the context hole with the corresponding hole type, and write $C[\cdot_\sigma]$ for the context with hole type σ. Thus, a context is well-typed iff it type checks with the extra rule:

$$\textsc{Ctxt}\ \frac{C[x] : \tau,}{C[\cdot_\sigma] : \tau}\ \text{where } x \text{ does not occur in } C \text{ and } \Gamma(x) = \sigma$$

Clearly, if $C[\cdot_\sigma] : \tau$ and $s : \sigma$, then $C[s] : \tau$.

For defining the standard reduction, we introduce A-, LR-, and R-contexts. The A-contextd are usual call-by-name evaluation contexts adapted to the constructs of the language (i.e. treating `fix`, `succ`, `pred`, and `if-then-else` as operators strict in their first argument) while LR-contexts represent an outer environment consisting of `let`-expressions. The reduction contexts R combine LR- and A-contexts such that the hole-position can go into the `in`-expressions of `let`, and also into a right-hand-side of a `let`-binding if it is needed – i.e. the bound variable occurs in a reduction context. The *flat A-contexts* A^1 are A-contexts with hole-depth 1

$$\begin{array}{rcl} A & ::= & [\cdot] \mid (A\ s) \mid \texttt{if}\ A\ \texttt{then}\ s\ \texttt{else}\ t \mid \texttt{pred}\ A \mid \texttt{succ}\ A \mid \texttt{fix}\ A \\ A^1 & ::= & ([\cdot]\ s) \mid \texttt{if}\ [\cdot]\ \texttt{then}\ s\ \texttt{else}\ t \mid \texttt{pred}\ [\cdot] \mid \texttt{succ}\ [\cdot] \mid \texttt{fix}\ [\cdot] \\ \mathrm{LR} & ::= & [\cdot] \mid \texttt{let}\ x = s\ \texttt{in}\ \mathrm{LR} \\ R & ::= & \mathrm{LR}[A] \mid \mathrm{LR}[\texttt{let}\ x = A\ \texttt{in}\ R[x]] \end{array}$$

A value in probabilistic call-by-need PCF is a natural or an abstraction. A weak head normal form additionally allows an outer LR-context.

$$\text{Values: } v ::= n \mid \lambda x.s \qquad\qquad \text{WHNFs: } w ::= \mathrm{LR}[v]$$

Standard reduction of probabilistic call-by-need PCF is a small-step reduction that respects sharing:

Definition 2.3 (Standard reduction)**.** The standard reduction of probabilistic call-by-need PCF is denoted by $\xrightarrow{sr}$ and defined by the following rules:

(sr,fix)	$R[\texttt{fix}\ \lambda x.s] \to R[(\lambda x.s)\ (\texttt{fix}\ \lambda x.s)]$
$(sr,lbeta)$	$R[(\lambda x.s)\ t] \to R[\texttt{let}\ x = t\ \texttt{in}\ s]$
$(sr,succ)$	$R[\texttt{succ}\ n] \to R[m]$ with $m = n+1$
$(sr,pred)$	$R[\texttt{pred}\ n] \to R[m]$ with $m = \max(0,n-1)$
$(sr,if\text{-}0)$	$R[\texttt{if}\ 0\ \texttt{then}\ s\ \texttt{else}\ t] \to R[s]$
$(sr,if\text{-}not\text{-}0)$	$R[\texttt{if}\ n\ \texttt{then}\ s\ \texttt{else}\ t] \to R[t]$ if $n \neq 0$
$(sr,probl)$	$R[s \oplus t] \to R[s]$
$(sr,probr)$	$R[s \oplus t] \to R[t]$
$(sr,lflata)$	$R[A^1[(\texttt{let}\ x = s\ \texttt{in}\ t)]] \to R[\texttt{let}\ x = s\ \texttt{in}\ A^1[t]]$
$(sr,llet)$	$\mathrm{LR}[\texttt{let}\ x = (\texttt{let}\ y = s\ \texttt{in}\ t)\ \texttt{in}\ R[x]] \to \mathrm{LR}[\texttt{let}\ y = s\ \texttt{in}\ \texttt{let}\ x = t\ \texttt{in}\ R[x]]$
(sr,cp)	$\mathrm{LR}[\texttt{let}\ x = v\ \texttt{in}\ R[x]] \to \mathrm{LR}[\texttt{let}\ x = v\ \texttt{in}\ R[v]]$

We apply standard reductions only to typed expressions. For a standard reduction step $s \xrightarrow{sr} t$, we sometimes write $s \xrightarrow{sr,lab} t$ where (sr,lab) is the name of the applied reduction rule, e.g. $s \xrightarrow{sr,cp} t$. With $\xrightarrow{sr,+}$ we denote the transitive closure of $\xrightarrow{sr}$, and with $\xrightarrow{sr,*}$ we denote the transitive-reflexive closure.

Rule (fix) evaluates the fixpoint-operator and requires the argument to be an abstraction, rule $(lbeta)$ evaluates the application of an abstraction to an argument, where the argument is shared by a new `let`-binding to avoid duplication of the argument. Rule (cp) copies a `let`-binding into a referenced position, but only if i) the expression is *needed*, and thus the referenced position is in a reduction context, and ii) the expression is already successfully evaluated, and thus it is a value (a natural number or an abstraction). If the expression is not a value, the reduction will evaluate the left-hand side of the `let`-binding (that is the reason why the construction of R-contexts is quite complicated). Rules $(succ)$ and $(pred)$ evaluate the successor and predecessor of a natural number, where `pred` 0 is evaluated to 0. Rules $(if\text{-}0)$ and $(if\text{-}not\text{-}0)$ evaluate the `if-then else`-expression. Rules $(lflata)$ and $(llet)$ adjust `let`-expression w.r.t. applications, `if-then-else`, `succ`, `pred`, `fix` and `let`-bindings by moving them over the corresponding construct. Rules $(probl)$ and $(probr)$ are the only source of non-determinism of the reduction, they evaluate a $\oplus$-expression by either choosing the left or the right argument. For now, this is completely non-deterministic, we will add probabilities to this decision in the next subsection. We say that a standard-reduction is a *prob*-reduction if it is a *probl*- or *probr*-reduction.

Note that for every typed expression, either no standard reduction is applicable (if s is a WHNF, or an open expression of the form $R[x]$), or s is of the form $R[r_1 \oplus r_2]$ and there are exactly two possibilities for the standard reduction: $s \xrightarrow{sr,probl} R[r_1]$ and $s \xrightarrow{sr,probr} R[r_2]$, or there is exactly one standard reduction applicable to s (i.e. $s \xrightarrow{sr} t$, but there is no t' with $s \xrightarrow{sr} t'$ and $t \neq t'$).

Definition 2.4 (Convergence)**.** For an expression s, we say that s *converges*, denoted as $s \downarrow w$, iff there is a reduction sequence $s \xrightarrow{*} w$, where w is a WHNF.

While for open expressions the reduction may get stuck, for closed well-typed expressions we have the following lemma (where we omit the proof):

Proposition 2.5 (Progress)**.** *If $s : \rho$ is closed and $s \not\xrightarrow{sr}$, then s is a (closed) WHNF.*

By inspecting the type rules together with the reduction rules, we obtain:

Proposition 2.6 (Type preservation)**.** *If $s:\rho$ and $s \xrightarrow{sr} t$. Then $t:\rho$.*

Definition 2.7. Let s be an expression. A sequence of standard reductions $s \xrightarrow{sr,*} t$ where t is a WHNF is called an *evaluation* of s. We write $Eval(s)$ for the set of all evaluations of expression s. For a reduction sequence $s \xrightarrow{sr,a_1} \cdots \xrightarrow{sr,a_n} t$, with $\mathsf{PS}(s \xrightarrow{sr,a_1} \cdots \xrightarrow{sr,a_n} t)$ we denote the subsequence of labels $a_i \ldots, a_k$ that are constructed from the sequence $a_1, \ldots, a_n$ by removing all labels a_j with $a_j \notin \{probl, probr\}$. We call the sequence the *prob-sequence* of the reduction sequence.

Since standard reduction has unique redexes and it is only non-deterministic for *prob*-reduction, the prob-sequence together with the start-expression s uniquely identifies an evaluation of s. Thus, to identify a single evaluation $s \xrightarrow{sr,*} t$ in the set $Eval(s)$, we write $s \wr_L t \in Eval(s)$, where $L = \mathsf{PS}(s \xrightarrow{sr,*} t)$.

Example 2.8. Let $f = (\lambda u.(0 \oplus \mathtt{succ}\ u))$ and let $g = \mathtt{fix}\ f$. To illustrate standard reduction, we show two evaluations of g (note that there are infinitely many):

$$g \xrightarrow{sr,fix} f\ (\mathtt{fix}\, f) \xrightarrow{sr,lbeta} \mathtt{let}\ u = \mathtt{fix}\, f\ \mathtt{in}\ (0 \oplus \mathtt{succ}\ u) \xrightarrow{sr,probl} \mathtt{let}\ u = \mathtt{fix}\, f\ \mathtt{in}\ 0$$

$$\begin{aligned} g &\xrightarrow{sr,fix} f\ (\mathtt{fix}\, f) \xrightarrow{sr,lbeta} \mathtt{let}\ u = \mathtt{fix}\, f\ \mathtt{in}\ (0 \oplus \mathtt{succ}\ u) \xrightarrow{sr,probr} \mathtt{let}\ u = \mathtt{fix}\, f\ \mathtt{in}\ \mathtt{succ}\ u \\ &\xrightarrow{sr,fix} \mathtt{let}\ u = f\ (\mathtt{fix}\, f)\ \mathtt{in}\ \mathtt{succ}\ u \xrightarrow{sr,lbeta} \mathtt{let}\ u = (\mathtt{let}\ u_1 = \mathtt{fix}\, f\ \mathtt{in}\ (0 \oplus \mathtt{succ}\ u_1))\ \mathtt{in}\ \mathtt{succ}\ u \\ &\xrightarrow{sr,llet} \mathtt{let}\ u_1 = \mathtt{fix}\, f\ \mathtt{in}\ \mathtt{let}\ u = (0 \oplus \mathtt{succ}\ u_1)\ \mathtt{in}\ \mathtt{succ}\ u \\ &\xrightarrow{sr,probl} \mathtt{let}\ u_1 = \mathtt{fix}\, f\ \mathtt{in}\ \mathtt{let}\ u = 0\ \mathtt{in}\ \mathtt{succ}\ u \xrightarrow{sr,cp} \mathtt{let}\ u_1 = \mathtt{fix}\, f\ \mathtt{in}\ \mathtt{let}\ u = 0\ \mathtt{in}\ \mathtt{succ}\ 0 \\ &\xrightarrow{sr,succ} \mathtt{let}\ u_1 = \mathtt{fix}\, f\ \mathtt{in}\ \mathtt{let}\ u = 0\ \mathtt{in}\ 1 \end{aligned}$$

For the first evaluation, the prob-sequence is *probl* and for the second one the prob-sequence is *probr*,*probl*. For the expression g, any prob-sequence of any evaluation is of the form $probr^*, probl$, since the evaluation ends by choosing the a left argument of the current $\oplus$-expression and it proceeds with the successor by choosing the right argument.

2.2 Weighted Reduction and Expected Convergence

Our operational semantics is non-deterministic, but it does not track the probability of different evaluations or results (in form of the WHNFs). To track the probabilities we introduce weights:

Definition 2.9 (Weighted Expressions and Reduction)**.** A *weighted expression* is a pair (p,s) where $p \in (0,1]$ is a rational number and s is an expression. Let (p,s) be a weighted expression. A *weighted standard reduction step* $\xrightarrow{wsr}$ (or $\xrightarrow{wsr,a}$ to make the rule explicit) on (p,s) is defined as follows:

$$(p,s) \xrightarrow{wsr,a} \begin{cases} (p,t) & \text{iff } s \xrightarrow{sr,a} t \text{ and } a \notin \{probl, probr\} \\ \left(\frac{p}{2}, t\right) & \text{iff } s \xrightarrow{sr,a} t \text{ and } a \in \{probl, probr\} \end{cases}$$

We use $\xrightarrow{wsr,*}$ and $\xrightarrow{wsr,+}$ for the reflexive-transitive, or transitive closure of $\xrightarrow{wsr}$.

An *evaluation* of (p,s) is a sequence $(p,s) \xrightarrow{wsr,*} (q,t)$ where t is a WHNF. We use the same notations as for non-weighted expressions. Thus, the sequence of labels a for each prob-reduction $\xrightarrow{wsr,a}$ together with the weighted expression (p,s) uniquely identifies an evaluation of (p,s)).

Example 2.10. We reconsider the expression g from Example 2.8, i.e. let $f = (\lambda u.(0 \oplus \mathtt{succ}\ u))$ and let $g = \mathtt{fix}\, f$. We show two weighted evaluations of $(1,g)$.

$$(1,g) \xrightarrow{wsr,fix} (1,f\,(\texttt{fix}\,f)) \xrightarrow{wsr,lbeta} (1,\texttt{let}\ u = \texttt{fix}\,f\ \texttt{in}\ (0 \oplus \texttt{succ}\ u)) \xrightarrow{wsr,probl} (\tfrac{1}{2},\texttt{let}\ u = \texttt{fix}\,f\ \texttt{in}\ 0)$$

$$\begin{aligned}(1,g) &\xrightarrow{wsr,fix} (1,f\ (\texttt{fix}\,f)) \xrightarrow{wsr,lbeta} (1,\texttt{let}\ u = \texttt{fix}\,f\ \texttt{in}\ (0 \oplus \texttt{succ}\ u)) \\ &\xrightarrow{wsr,probr} (\tfrac{1}{2},\texttt{let}\ u = \texttt{fix}\,f\ \texttt{in}\ \texttt{succ}\ u) \xrightarrow{wsr,fix} (\tfrac{1}{2},\texttt{let}\ u = f\ \texttt{fix}\,f\ \texttt{in}\ \texttt{succ}\ u) \\ &\xrightarrow{wsr,lbeta} (\tfrac{1}{2},\texttt{let}\ u = (\texttt{let}\ u_1 = \texttt{fix}\,f\ \texttt{in}\ (0 \oplus \texttt{succ}\ u_1))\ \texttt{in}\ \texttt{succ}\ u) \\ &\xrightarrow{wsr,llet} (\tfrac{1}{2},\texttt{let}\ u_1 = \texttt{fix}\,f\ \texttt{in}\ \texttt{let}\ u = (0 \oplus \texttt{succ}\ u_1)\ \texttt{in}\ \texttt{succ}\ u) \\ &\xrightarrow{wsr,probl} (\tfrac{1}{4},\texttt{let}\ u_1 = \texttt{fix}\,f\ \texttt{in}\ \texttt{let}\ u = 0\ \texttt{in}\ \texttt{succ}\ u) \\ &\xrightarrow{wsr,cp} (\tfrac{1}{4},\texttt{let}\ u_1 = \texttt{fix}\,f\ \texttt{in}\ \texttt{let}\ u = 0\ \texttt{in}\ \texttt{succ}\ 0) \xrightarrow{wsr,succ} (\tfrac{1}{4},\texttt{let}\ u_1 = \texttt{fix}\,f\ \texttt{in}\ \texttt{let}\ u = 0\ \texttt{in}\ 1)\end{aligned}$$

Definition 2.11 (Expected Convergence)**.** For a weighted expression (p,s) we denote with $Eval(p,s)$ the set of evaluations of (p,s), where single evaluations in the set are written $(p,s)\searrow_L(q,t)$, i.e. if $(p,s)\searrow_L(q,t) \in Eval(p,s)$ then there is an evaluation $(p,s) \xrightarrow{wsr,*} (q,t)$ where (q,t) is the resulting weighted WHNF and $L = \mathsf{PS}(s \xrightarrow{sr,*} t)$. The *expected convergence* $\textsc{ExCv}(s)$ of an expression s is the (perhaps infinite) sum of all weights of WHNFs that are reachable from $(1,s)$, i.e.

$$\textsc{ExCv}(s) = \sum_{(1,s)\searrow_L(q,t)\in Eval(1,s)} q.$$

For a number n, the *expected value convergence of s on value n* $\textsc{ExVCv}(s,n)$ is the (infinite) sum:

$$\textsc{ExVCv}(s,n) = \sum_{(1,s)\searrow_L(q,\mathrm{LR}[n])\in Eval(1,s)} q,$$

i.e. only evaluations resulting in expressions of the form $\mathrm{LR}[n]$ (where n is a number) are taken into account. In contrast in $\textsc{ExCv}(s)$ any WHNF t is taken into account as a result.

The notions of expected convergence and also expected value-convergence are well-defined, that is, the limit $\sum_{(1,s)\searrow_L(q,t)\in Eval(1,s)} q$ always exists and is unique. In particular, its value is independent of the enumeration of the countable set $Eval(s)$. For a rigorous proof, in another calculus, see [16].

The following lemma holds, since both expressions of the equation define the same infinite sum:

Lemma 2.12. *For all expressions $s : nat$:* $\textsc{ExCv}(s) = \sum_{i=0}^{\infty} \textsc{ExVCv}(s,i)$

Example 2.13. For $s = (\lambda x.((\texttt{succ}\ x) \oplus (\texttt{pred}\ x)))\ (0 \oplus 1)$, the expected convergence is $\textsc{ExCv}(s) = 1$. The expected value convergences are $\textsc{ExVCv}(s,0) = \frac{1}{2}$, $\textsc{ExVCv}(s,1) = \frac{1}{4}$, $\textsc{ExVCv}(s,2) = \frac{1}{4}$ and $\textsc{ExVCv}(s,i) = 0$ for all $i > 2$.

3 Equalities

Contextual preorder and equivalence observe the expected convergence of expressions in any program context. For the preorder, the expected convergence is never decreased and for contextual equivalence, the expected convergence is the same for both expressions inside any context.

Definition 3.1 (Contextual Preorder and Equivalence)**.** Let $s,t :: \sigma$ be equally typed expressions. If for all contexts $C[\cdot_\sigma] : nat$, we have $\textsc{ExCv}(C[s]) \le \textsc{ExCv}(C[t])$, then $s \le_c t$. We define $s \sim_c t$ iff $s \le_c t$ and $t \le_c s$. The relation $\le_c$ is called *contextual preorder*, and $\sim_c$ is called *contextual equivalence*.

Example 3.2. Refuting an equivalence $s \sim_c t$ (and thus showing $s \not\sim_c t$) can be done by a single context C that acts as a counter-example, i.e. $\textsc{ExCv}(C[s]) \neq \textsc{ExCv}(C[t])$. For instance, for $n,m \in \mathbb{N}_0$ with $n < m$ and the context $C := \texttt{if}\ \texttt{pred}^n\ [\cdot]\ \texttt{then}\ 0\ \texttt{else}\ (\texttt{fix}\ \lambda x.x)$ (where $\texttt{pred}^n$ is the n-fold application of $\texttt{pred}$) we have $\textsc{ExCv}(C[n]) = 1$ but $\textsc{ExCv}(C[m]) = 0$, since $\texttt{fix}\ \lambda x.x$ diverges.

It is straightforward to show that $\sim_c$ is an equivalence relation and that $\leq_c$ is a preorder. Since both relations are compatible with contexts, we have:

Proposition 3.3. *The relation $\sim$ is a congruence, and the relation $\leq_c$ is a precongruence.*

Remark 3.4. Call-by-value and call-by-name variants of our language would have different operational and also contextual semantics. For instance, in a call-by-value language $(\lambda y.1)\ \bot$ is contextually equivalent to $\bot$, where $\bot$ is a non-terminating expression, for example $(\texttt{fix}\ \lambda x.x)$. In *ProbPCF*^{need} the expressions are not equivalent, since the former expressions evaluates to the WHNF $\texttt{let}\ y = \bot\ \texttt{in}\ 1$ and the latter diverges. In deterministic languages, call-by-need evaluation can be seen as an efficient implementation of the call-by-name semantics. In our probabilistic (and hence non-deterministic) setting, this is no longer true, since the semantic changes: for instance, let *add* be a function implementing addition, then $((\lambda\ x.add\ x\ x)\ (1 \oplus 2)$ evaluates in call-by-name to $2,3,4$ (where 2 and 4 have probability $\frac{1}{4}$, and 3 has probability $\frac{1}{2}$) while in *ProbPCF*^{need} only the results 2 and 4 are possible (both with probability $\frac{1}{2}$, since the argument $(1 \oplus 2)$ is evaluated only once and the result is shared for the computation of $(add\ x\ x)$.

Remark 3.5. Contextual equivalence would slightly change if also convergence on other types like function types is observed. An example are the expressions $\lambda x.\bot$ and $\bot$. These cannot be distinguished, since in the case the reduction uses one of these expressions, evaluation does not terminate. This is an indication that the η-law may hold in our formulation of the contextual equivalence. Thus the reduction strategy is slightly different to the ones in lazy functional programming languages like Haskell [5] where a `seq`-operator is available to enforce strict evaluation of any expression.

Showing that two expressions are contextually equivalent requires to reason about any context C and thus it is harder than refuting equivalences and requires additional tools and techniques. In the next section we show that a context lemma holds which is an important tool.

3.1 Context Lemma

For an expression s, we define $\text{ExCv}(s,k)$ as the (perhaps infinite) sum of all weights of WHNFs that are reachable from $(1,s)$ where at most k prob-reductions are used (in contrast, in $\text{ExCv}(s)$ any number of prob-reductions is allowed). For instance, for expression g from Examples 2.8 and 2.10, we have $\text{ExCv}(g) = 1$, but $\text{ExCv}(g,0) = 0$ (since at least one prob-reduction is necessary to reach a WHNF), $\text{ExCv}(g,1) = \frac{1}{2}$ (since with one prob-reduction only one WHNF is reachable), $\text{ExCv}(g,2) = \frac{3}{4}$ (since with two prob-reductions a further WHNF is reachable), $\text{ExCv}(g,3) = \frac{7}{8}$, asf.

While the bound on the number k is helpful for inductive proofs (since one can perform induction on this number), the following lemma shows how to transfer results with bounds to the limit: it shows that preservation of expected convergence for any bound on the prob-length implies preservation of expected convergence. The additional parameter d allows some difference between the evaluations of s and t.

Lemma 3.6. *Let $s,t:\tau$ such that $\forall k \geq 0 : \exists d : \text{ExCv}(s,k) \leq \text{ExCv}(t,k+d)$. Then $\text{ExCv}(s) \leq \text{ExCv}(t)$.*

The proof is similar to [16, Lemma 3.3] for a different probabilistic lambda-calculus.

Now we can state the context lemma for *ProbPCF*^{need}. It is proved for a related language in [16], and the proof can be transferred to our setting. Here we use multi-contexts, i.e. expressions with several (or no) holes. We only consider multi-contexts where all holes are of the same type. Thus, we write $C[[\cdot_{1,\sigma}],\ldots,[\cdot_{N,\sigma}]]$ for a multi-context with $N \in \mathbb{N}_0$ holes – all of type σ. A multi-context is well-typed with type ρ iff $C[x,\ldots,x]:\rho$ for fresh variable x with $\Gamma(x) = \sigma$.

Theorem 3.7 (Context Lemma)**.** *Let $N \geq 0$, let σ be a type and for $1 \leq i \leq N$, let s_i,t_i be expressions of type σ, such that $\forall k \geq 0$, $\forall$ reduction contexts $R[\cdot_\sigma]$, of type $\sigma \to nat$ there exists $d \geq 0$:*

(*fix*)	$\texttt{fix}\ \lambda x.s \to (\lambda x.s)\ (\texttt{fix}\ \lambda x.s)$	(*if-not*-0)	$\texttt{if}\ n\ \texttt{then}\ s\ \texttt{else}\ t \to t$ if $n \neq 0$
(*lbeta*)	$((\lambda x.s)\ t) \to \texttt{let}\ x = t\ \texttt{in}\ s$	(*lflata*)	$A^1[(\texttt{let}\ x = s\ \texttt{in}\ t)] \to \texttt{let}\ x = s\ \texttt{in}\ A^1[t]$
(*succ*)	$(\texttt{succ}\ n) \to m$ with $m = n+1$	(*llet*)	$\texttt{let}\ x = (\texttt{let}\ y = s\ \texttt{in}\ t)\ \texttt{in}\ r$
(*pred*)	$(\texttt{pred}\ n) \to m$ with $m = \max(0, n-1)$		$\to \texttt{let}\ y = s, x = t\ \texttt{in}\ r$
(*if*-0)	$\texttt{if}\ 0\ \texttt{then}\ s\ \texttt{else}\ t \to s$	(*cp*)	$\texttt{let}\ x = v\ \texttt{in}\ C[x] \to \texttt{let}\ x = v\ \texttt{in}\ C[v]$

Figure 2: Program Transformations

$\text{ExCv}(R[s_i], k) \le \text{ExCv}(R[t_i], k+d)$. *Let* $C[\cdot_{1,\sigma}, \ldots, \cdot_{N,\sigma}] : nat$ *be a multicontext with N holes. Then the inequation* $\text{ExCv}(C[s_1, \ldots, s_N]) \le \text{ExCv}(C[t_1, \ldots, t_N])$ *holds.*

In the proof of the context lemma, one first shows that for all $k \ge 0$ there exists a d such that $\text{ExCv}(C[s_1, \ldots, s_N], k) \le \text{ExCv}(C[t_1, \ldots, t_N], k+d)$. Then Lemma 3.6 is applied in a final step. For the former a part, the proof is by induction on the lexicographically ordered) tripel (k, l, N) where k is the bound on the number of prob-reductions, l is the maximal length of the evaluations of $C[s_1, \ldots, s_n]$ that use at most k prob-reductions, and N is the number of holes in C.

3.2 Program Transformations

A program transformation $\xrightarrow{T}$ is a binary relation on equally typed expressions, it is *correct* if it does not change the semantics of a program. Since in our setting programs are semantically equal iff they are contextual equivalent, we define correctness of a program transformation by $\xrightarrow{T} \subseteq \sim_c$. The context lemma will be very helpful in arguing on the correctness of reduction rules as program transformations. In Fig. 2 generalisations of standard reductions are shown.

Proposition 3.8. *The context lemma immediately implies that all of the transformations (fix), (lbeta), (succ), (pred), (if*-0*), (if-not*-0*), (lflata) are correct.*

Proof. This holds, since for any reduction context R and any of the formerly mentioned transformations $s \xrightarrow{T} t$, $R[s] \to R[t]$ is always a standard reduction step (that is not a prob-reduction). Thus we have that $\text{ExCv}(R[s], k) = m$ iff $\text{ExCv}(R[t], k) = m$. This shows the precondition of the context lemma for $N = 1$ (and where d is always 0). The context lemma shows that $s \sim_c t$ holds. □

Proving correctness of the transformations (*llet*) and (*cp*) requires more sophisticated techniques, but we are convinced that their correctness can be proved in a similar way as we proved correctness for similar transformations in [16]. However, we leave the rigorous proof as future work.

Also transformations related to the $\oplus$-operator can be shown correct. For instance, commutativity $(s \oplus t) \to (t \oplus s)$ can be shown correct by using the context lemma and the observation that $R[s \oplus t]$ standard reduces to $R[s]$ or $R[t]$ (both with probability $\frac{1}{2}$) and the same holds for $(t \oplus s)$. Also the law $(s \oplus s) \to s$ is a correct program transformation, that can be shown correct by the context lemma.

The associativity law $(r \oplus (s \oplus t)) \to ((r \oplus s) \oplus t)$ is not correct, since probabilities are changed. However, the adjusted (distributivity) transformation $(r \oplus (s \oplus t)) \to ((r \oplus s) \oplus (r \oplus t))$ is correct: for the left expression and any reduction context R:

$$\text{ExCv}(R[(r \oplus (s \oplus t))], k) = \frac{1}{2}\text{ExCv}(R[r], k-1) + \frac{1}{4}\text{ExCv}(R[s], k-2) + \frac{1}{4}\text{ExCv}(R[t], k-2).$$

For the right expression and any reduction context R:

$$\text{ExCv}(R[((r\oplus s)\oplus(r\oplus t))],k) = 2\cdot\frac{1}{4}\text{ExCv}(R[r],k-2)+\frac{1}{4}\text{ExCv}(R[s],k-2)+\frac{1}{4}\text{ExCv}(R[t],k-2).$$

Since $\text{ExCv}(s_0,k)\geq\text{ExCv}(s_0,k')$ for any expression s_0 and $k\geq k'$, the equations imply

$$\begin{array}{ll}\text{ExCv}(R[((r\oplus s)\oplus(r\oplus t))],k) & \leq \text{ExCv}(R[(r\oplus(s\oplus t))],k) \quad \text{and}\\ \text{ExCv}(R[(r\oplus(s\oplus t))],k) & \leq \text{ExCv}(R[((r\oplus s)\oplus(r\oplus t))],k+1)\end{array}$$

for all $k\geq 0$. The context lemma for $N=1$ holes shows contextual equivalence of the expressions. Also, the transformation $((r_1\oplus s_1)\oplus(r_2\oplus s_2))\to((r_1\oplus s_2)\oplus(r_2\oplus s_1))$, is correct, which again can be proved by the context lemma. In [16], worked-out proofs for a related (untyped) calculus can be found.

A transformation proved in [19] for a strict probabilistic programming language with let is $\texttt{let } x = t_x \texttt{in let } y = t_y \texttt{ in } s \to \texttt{let } x = t_x \texttt{ in let } y = t_y \texttt{ in } s$ if x does not occur free in t_y. We conjecture that this transformation is also correct in *ProbPCF*^{need}. The context-lemma together with checking the overlaps appears to be sufficient to show equivalence. However, in a call-by-value language, the above transformation changes the order of the effects that appear while evaluating t_x and t_y, while in our call-by-need language, the order is the same (whether t_x and/or t_y are evaluated and the order depends on whether and/or when the evaluation of s demands the value of x and y).

3.3 Distribution-Equivalence

We are also interested in the following operational equality of expressions that identifies expressions as equal iff they generate the same probability distribution on natural numbers:

Definition 3.9 (Distribution-Equivalence)**.** Let $s,t:nat$ be two closed expressions. Then s and t are *distribution-equivalent*, written $s\sim_d t$, iff for all $n\in\mathbb{N}_0$: $\text{ExVCv}(s,n)=\text{ExVCv}(t,n)$.

Example 3.10. We reconsider Example 2.8, i.e. $g=\texttt{fix } f$ where $f=(\lambda u.(0\oplus\texttt{succ } u))$. The expression g produces natural numbers i with probability $\frac{1}{2^{i+1}}$, i.e. g produces 0 with probability $\frac{1}{2}$, 1 with probability $\frac{1}{4}$ asf. We claim that this sequence, even if obtained through an infinite sum, is characteristic for contextual equivalence. Another example is $(0\oplus 1)+2*(0\oplus 1)$ and $(0\oplus 1)\oplus(2\oplus 3)$ where $+$ and $*$ are encodings of addition and multiplication. The first expression corresponds to throwing sequentially 2 coins, and the second to throwing a 4-sided dice with probability 0.25. Both expressions are distribution-equivalent.

A result on the contextual equivalences is:

Theorem 3.11. *Let $s,t:\sigma$ be two typed expressions with $s\sim_c t$. Then for any context $C[\cdot_\sigma]:nat$, the generated distribution is the same for $C[s]$ and $C[t]$, i.e. $C[s]\sim_d C[t]$.*

Proof. Let n be some number, and let C_n be a context that recognizes the number n, i.e. $C_n[\cdot_{nat}] = \texttt{if } [\cdot_{nat}] == n \texttt{ then } 0 \texttt{ else } bot$ (where *bot* is an expression of type *nat* that diverges, e.g. $bot=\texttt{fix } \lambda x.x$ with $\Gamma(x)=nat$). The context C_n can be programmed in *ProbPCF*^{need}. The context $C_n[C[\cdot_\sigma]]$ is now used to extract the probabilities. We know that $C_n[C[s]]\sim_c C_n[C[t]]$, and more detailed, it follows that $\text{ExVCv}(C_n[C[s]])$ and $\text{ExVCv}(C_n[C[t]])$ are exactly the respective probabilities $\text{ExVCv}(s,n)$ and $\text{ExVCv}(t,n)$, which must be equal since s,t are contextually equivalent. □

The issue is whether the converse holds. We do not yet have a proof but there is some hope that the method in [7], which was already extended to non-deterministic call-by-need calculi with let in [12] can be adapted. Thus we formulate a conjecture:

Conjecture 3.12. *If the distribution of closed expressions $s,t:nat$ in the empty context is the same (i.e. $s\sim_d t$), then s,t are contextually equivalent.*

4 Conclusion

We introduced a monomorphically typed call-by-need functional language with fair probabilistic choice. We defined contextual equivalence and distribution-equivalence. Besides showing a context lemma and correctness of program transformations, we showed that contextual equivalence implies distribution-equivalence. For the opposite direction, we formulated a conjecture and leave its proof as future work.

References

[1] Z. M. Ariola and M. Felleisen. The call-by-need lambda calculus. *JFP*, 7(3):265–301, 1997.

[2] Z. M. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. A call-by-need lambda calculus. In *POPL 1995*, pages 233–246. ACM, 1995. doi:10.1145/199448.199507.

[3] R. Crubillé and U. Dal Lago. On probabilistic applicative bisimulation and call-by-value λ-calculi. In *ESOP 2014*, volume 8410 of *LNCS*, pages 209–228. Springer, 2014. doi:10.1007/978-3-642-54833-8_12.

[4] A. Silva G. Barthe, J.-P. Katoen, editor. *Foundations of Probabilistic Programming*. Cambridge University Press, 2020. doi:10.1017/9781108770750.

[5] Haskell. Haskell, an advanced, purely functional programming language, 2022. URL: www.haskell.org.

[6] B. T. Howard and J. C. Mitchell. Operational and axiomatic semantics of PCF. In *LFP 1990*, pages 298–306. ACM, 1990. doi:10.1145/91556.91677.

[7] D. J. Howe. Equality in lazy computation systems. In *LICS 1989*, pages 198–203. IEEE Computer Society, 1989. doi:10.1109/LICS.1989.39174.

[8] C. Jones and G. D. Plotkin. A probabilistic powerdomain of evaluations. In *LICS 1989*, pages 186–195. IEEE Computer Society, 1989. doi:10.1109/LICS.1989.39173.

[9] U. Dal Lago. *On Probabilistic Λ-Calculi*, pages 121–144. Cambridge University Press, 2020. doi:10.1017/9781108770750.005.

[10] U. Dal Lago, D. Sangiorgi, and M. Alberti. On coinductive equivalences for higher-order probabilistic functional programs. In *POPL 2014*, pages 297–308. ACM, 2014. doi:10.1145/2535838.2535872.

[11] U. Dal Lago and M. Zorzi. Probabilistic operational semantics for the lambda calculus. *RAIRO-Theor. Inf. Appl.*, 46(3):413–450, 2012. doi:10.1051/ita/2012012.

[12] M. Mann and M. Schmidt-Schauß. Similarity implies equivalence in a class of non-deterministic call-by-need lambda calculi. *Inf. Comput.*, 208(3):276–291, 2010. doi:10.1016/j.ic.2009.11.003.

[13] I. A. Mason, S. F. Smith, and C. L. Talcott. From operational semantics to domain theory. *Inf. Comput.*, 128(1):26–47, 1996. doi:10.1006/inco.1996.0061.

[14] J. H. Morris. *Lambda-Calculus Models of Programming Languages*. PhD thesis, MIT, 1968.

[15] G. D. Plotkin. LCF considered as a programming language. *Theor. Comput. Sci.*, 5(3):223–255, 1977. doi:10.1016/0304-3975(77)90044-5.

[16] D. Sabel, M. Schmidt-Schauß, and L. Maio. A probabilistic call-by-need lambda-calculus – extended version. *CoRR*, 2022. doi:10.48550/ARXIV.2205.14916.

[17] N. Saheb-Djahromi. Probabilistic LCF. In *MFCS 1978*, volume 64 of *LNCS*, pages 442–451. Springer, 1978. doi:10.1007/3-540-08921-7_92.

[18] M. Schmidt-Schauß and D. Sabel. On generic context lemmas for higher-order calculi with sharing. *Theoret. Comput. Sci.*, 411(11-13):1521 – 1541, 2010. doi:10.1016/j.tcs.2009.12.001.

[19] M. Wand, R. Culpepper, T. Giannakopoulos, and A. Cobb. Contextual equivalence for a probabilistic language with continuous random variables and recursion. *Proc. ACM Program. Lang.*, 2(ICFP), 2018. doi:10.1145/3236782.