GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Department of Computer Science, Goethe-Universität Frankfurt am Main

Functional Programming

winter semester 2022/2023

Supervisor:
Prof. Dr. Manfred Schmidt-Schauß

Computing multi-distributions from probabilistic call-by-need functional programs

Nick Wagner

Frankfurt am Main
March 2, 2023

Acknowledgement

First of all, I would like to thank my thesis supervisor Prof. Dr. Manfred Schmidt-Schauß from the Chair of Artificial Intelligence and Software Technology at the Johann Wolfgang Goethe University for supervising me during my master's thesis. He has my deepest appreciation for the effort he has put into mentoring me, that he invested a lot of time in giving me tips, for the proofreading, suggestions and ideas and was always open to my questions and answered them when I ran into difficulties.

Finally, I must express my deepest gratitude to my parents and my friends for providing me with constant support and encouragement throughout my years of study and through the process of researching and writing this thesis and also for proofreading. This accomplishment would not have been possible without them. Thank you.

Statement of originality

I hereby confirm that I have written the accompanying thesis by myself, without contributions from any sources other than those cited in the text, references and acknowledgements. This applies explicitly to all graphics, drawings, maps, program code and images included in the thesis.

Contents

1 Introduction

In recent years, research of probabilism in functional programmings has grown up. Functional programming languages usually apply reductions to the program as long as the program evaluates to a value. The result depends on the evaluation strategy that the programming language takes into account and that is applied to the input. Due to the quantization of all possible strategies that exists, there may an infinite set of results, which makes reasoning a hard problem. In the literature, a common solution to this problem is, to fix the strategy which is also known as strict evaluation. While call-by-value and call-by-name are frequently researched strategies, this thesis attempts to study the call-by-need strategy, which often is preferable when the performance is paramount.
Most functional programming languages use monadic functions as a foundation for random-based computations. This makes perfect sense in some programming languages like Haskell, where monads are implemented directly [1]. However, there is a much lower-level approach to enabling probability calculus, which is to integrate probability calculus into the core of the programming language by extending the lambda calculus with a probabilistic choice operator. We investigate two versions of this operator. The first one models flipping a fair coin, by choosing either the first or the second argument with a 50/50 chance. The second offers the possibility of influencing the probabilities of the selection via a value. Given such an operator, the evaluation of probabilistic expressions is not longer a single deterministic value, rather than a distribution over all possible values.
Although the call-by-need strategy and the prob operator have been explored in many articles, the combination of the strategy and the extension is the current subject of research. The work is about improving the methods for computing distributions to provide a much more detailed analysis of the programs and to create a basis for later developments within functional programming, such as those required for AIs.

1.1 Related Work

The probabilistic call-by-need calculus Λ_{PNeedR} is the subject of current research. This calculus extends the classical calculus of Alonzo Church with recursive let expressions that enable sharing for call-by-need evaluation and a probabilistic operator $\oplus_p$.
The article [2] deals with examining fundamental properties for that calculus that includes contextual equality via the convergence behavior. Some program transformations are shown and examined for correctness. Success can be recorded by proving a so-called context lemma.
The most recent work [3] deals with the distribution equality of expressions in PCF, a probabilistic call-by-need calculus that uses Church Numerals as answers. This article shows that contextual and distributional equivalence coincide for closed expressions in PCF.
A very closely related work, the bachelor thesis [4] shows, that it is not trivial to compute the multi-distribution of a given program. A program may not stop evaluating due to non-termination properties. The programming language Λ_{PneedR}, which is the notation used in this thesis as well, is defined given a probabilistic lambda calculus that with a set of reduction rules that integrate recursive let expressions. In the article, multiple variations of contextually equality within the probabilistic setup are worked out. The work uses the Monte Carlo method to approximate the distributions of various expressions. This is tested by a Haskell interpreter that implements the reduction of expressions combined with weighted expressions.

1.2 Motivation

The computability of distributions for the lambda calculus Λ_{PNeedR} is not a trivial undertaking. Preliminary work has been done in [4], but in many cases it is doomed to failure. It must be mentioned that not every program has an evaluation, which can be related to the halting problem. Some expressions cannot be evaluated with the operational semantics used in [4], although they have a simple result that can be obtained even by intuition. One of the reasons for the failure of the calculations is recursion, which can result in a non-finite evaluation. The assumption that programs are equal if their distributions are equal, called distributional equivalence, opens up new possibilities. Program transformations can be designed that are justified based on their distributional equivalence, meaning that programs with possibly non-finite evaluation can be replaced by distributional equivalent programs. This provides a much deeper and more accurate analysis, since probabilities are directly included in the evaluation process. The predictability of the distributions is also an important aspect for deciding on the equality of programs, which is in causality closely related to optimization.

1.3 Overview

Foremost, this thesis provides an introduction to the terms of program calculi, especially the lambda calculi Λ_{LNeedR} and its simplification Λ_{LNeed} together with a set of reduction rules that implements sharing and call-by-need evaluation to complete the definition of a lazy programming language (chapter 2). It also introduces contexts and the reduction strategy, namely standard reduction and program transformations in the form of operational semantics. It becomes clear that the strategy fails for some expressions in the probabilistic setup. Chapter 3 deals with variations of equivalence. Two variants are mainly used throughout this work: Distribution and contextual equivalence. Chapter 4 introduces probability distributions and points out, how they are computed. We introduce a set of distribution rules to extract linear equations out of expressions. In chapter 5 we use the previously defined rules to analyze the recursive behavior of the expressions. It can be shown, that for some recursive programs, an evaluation in the notation of distribution can be found, where the evaluation with the standard reduction will fail. This can be done by extracting linear equations and dependent graphs from the expressions. The chapters 6 and 7 explain two approaches of solving the linear equations using Markov chains and Gauss Elimination. Since the solution of these strategies is given by distributions, chapter 8 explains an algorithm, that transforms distributions back into distribution equivalent expression of the calculus. Last but not least, Chapter 9 provides an explanation of a Haskell interpreter. The interpreter is a demonstration of the methods developed so far and shows their applicability in practice. A short discussion completes the thesis.

2 Program Calculus Λ_{PNeedR}

The definition of pure functional programming languages is given by a program calculus that is a 5-tuple $(\mathcal{E}, \mathcal{C}, \xrightarrow{sr}, \mathcal{A}, \mathcal{L})$, where $\mathcal{E}$ is the set of all expressions, $\mathcal{C}$ are *contexts* that specifies the strategy of evaluation, $\xrightarrow{sr} \subseteq \mathcal{E} \times \mathcal{E} \times \mathcal{L}$ the *reduction relation* which defines rewriting rules and simplification for expressions, $\mathcal{A}$ the set of *answers*, sometimes are called *values*, which are expressions that can not be evaluated further, and finally $\mathcal{L}$, the set of *labels*, which are names assigned to each rewrite or reduction rule [5]. To implement the call-by-need strategy and a probabilistic choice, the lambda-calculus Λ_{PNeedR} that was introduced in [2] is chosen. This lambda-calculus is an expansion of Alonzo Church's lambda calculus. It extends the calculus by the fair binary probabilistic choice operator $\oplus_p$ and a recursive let, like the one that the programming language Haskell provides. This enables the implementation of the call-by-need strategy with lazy evaluation and sharing. The definition is done by specifying the 5-tuple. While $\mathcal{E}$ and $\mathcal{C}$ typically can be denoted as a Backus-Naur-Form (BNF), the reduction rules, as well as answers and labels can simply be easily listed.

2.1 Expressions $\mathcal{E}$

The expressions of the calculus are defined by the given BNF:

$$s, t \in Expr ::= x \mid \lambda x.s \mid (s\ t) \mid \text{let } Env \text{ in } s \mid s \oplus_p t$$
$$Env ::= x_1 = s_1, ..., x_n = s_n$$

Let Var be an infinite, countable set of variables. We denote variables with the letters $x,\ y \in Var$, expressions with the letters $s,\ t \in Expr$. An expression can either be a *variable* (x), an *abstraction* ($\lambda x.s$), an *application* ($s\ t$), a *let-expression* (let Env in s), or a *prob-expression* ($s \oplus_p t$). The latter takes use of the infix operator $\oplus_p$ that is introduced in [4]. This operator evaluates either the left argument with probability p or the right argument with probability $1 - p$. [2] showed, that the operator is not associative, but distributive and idempotent w.r.t. contextual equivalence. The let-expressions contain an *environment* Env which is an unordered set $[x_i = s_i | i \in I]$ of assignments called *definitions*, that are in the scope of the in-expression. For such an environment, the variables x_i are called *left variables* LV. To uniquely identify the definitions, they are provided with an index from index set I and must be distinct. The definitions enable the lazy evaluation strategy by storing results that can be shared when needed.

2.2 Context $\mathcal{C}$

A general context C is an expression with exactly on hole $[\cdot]$, that is a placeholder for a sub-expression that can be plugged into. By the definition, usually given as a BNF, the possible position of the sub-expression are determined. This is an important feature during the process of reduction, where contexts are used to show the position of the next *redex* (short for reducible expression). Hence, they are the means of choice for specifying the evaluation strategy. Given a context, an expression s can be substituted into the hole. This is denoted by $C[s]$. We investigate *general contexts* $\mathbb{C}$, *reduction contexts* $\mathbb{R}$ and *application contexts* $\mathbb{A}$ by giving their BNF. These are required multiple times in this thesis.

$$\begin{aligned}
C \in \mathbb{C} \quad &::= [\cdot] \mid \lambda x.C \mid (C\ t) \mid (t\ C) \mid (C \oplus t) \mid (t \oplus C) \mid \text{let } env \text{ in } C \mid \text{let } env, y = C \text{ in } t \\
R \in \mathbb{R} \quad &::= A \mid \text{let } env \text{ in } A \mid \text{let } env, \{x_i = A_i[x_{i+1}]\}_{i=1}^{n}, x_{n+1} = A_{nj+1} \text{ in } t \\
A \in \mathbb{A} \quad &::= [\cdot] \mid (A\ s)
\end{aligned}$$

Reduction contexts show the positions of the redexes during standard reduction and, thus, define the lazy call-by-need strategy. Reducing the sub-expressions of reduction contexts suffices to evaluate an expression and to conclude contextual equivalence, like [2] has shown in the context lemma. General contexts mark every position that is a sub-expression. They are also used to define contextual equality.

2.3 Answers $\mathcal{A}$

Answers are expressions $\mathcal{A} \in \mathcal{E}$, that are used to encode types. Given a list of answers, the redexes of a reduction context are compared with the answers. Whenever the redex matches an answer, the expression is assumed to be fully reduced. An example is the *church encoding*, which encodes algebraic types throughout the lambda-calculus. Answers can be Church Numerals, True and False etc. encoded in expressions, as they are used in [3]. This thesis uses an untyped calculus with open expressions, since this is more general. Nevertheless, there is a set of expression that is naturally a subset of the answers, WHNFs namely.

2.3.1 WHNF

In general, a WHNF is an expression of a shape such that no reduction rule is applicable. For this reason, it is a subset of $\mathcal{A}$ In calculi that do not implement sharing and lazy evaluation, this is the case for abstractions of the form $\lambda x.s$. In our case, the term of WHNF must be adopted to the domain of Λ_{PNeed}, since it is enlarged by let-bindings that enables the implementation of sharing. It follows that expressions of form (let env in $\lambda x.s$) i.e. let-expression where the inner expression is an abstraction are not reducible and belong to WHNF as well. In the untyped calculus, a program terminates or evaluates successfully, if it reduces to a WHNF in a finite amount of standard reductions. In the other case, the sequence of standard reduction is transfinite. Due to the nondeterminism of the prob operator, an expression may have multiple reduction sequences that lead to various results. We call set of all results that are WHNFs, i.e. there is a standard reduction sequence $s \to \ldots \to$ WHNF the *evaluation* of s denoted by $Eval(s)$ [2].

2.4 Reduction Relation $\xrightarrow{sr}$

Reduction relations $\xrightarrow{sr} \subseteq \mathcal{E} \times \mathcal{E} \times \mathcal{L}$ are used to step-wise simplify expressions and thus define the operational, small-step semantics. For λ_{PNeed} we use the standard reduction rules of figure 2.1 as laid out in [4]. The set of reduction relations rules must be complete, which means that for all arbitrary expressions not in WHNF, there exists a relation rule applicable. It follows that when no standard reduction is applicable, the expression must be in WHNF. The labels $\mathcal{L}$ are provided by the brackets to the left.

(sr,lbeta)	$R[((\lambda x.s)t)] \to R[\text{let } x = t \text{ in } s]$
(sr,probl)	$R[s \oplus t] \to R[s]$
(sr,probr)	$R[s \oplus t] \to R[t]$
(sr,lapp)	$R[((\text{let } env \text{ in } s)\ t)] \to R[\text{let } env \text{ in } (s\ t)]$
(sr,llet-in)	$\text{let } env_1 \text{ in let } env_2 \text{ in } s \to \text{let } env_1, env2 \text{ in } s$
(sr,llet-e)	$\text{let } \{x_i = A_i[x_{i+1}]\}_{i=1}^{n-1}, x_n = (\text{let } env_1 \text{ in } s), env_2 \text{ in } A[x_1]$
	$\to \text{let } \{x_i = A_i[x_{i+1}]\}_{i=1}^{n-1}, x_n = s, env_1, env_2 \text{ in } A[x_1]$
(sr,cp-in)	$\text{let } \{x_i = x_{i+1}\}_{i=1}^{n-1}, x_n = \lambda y.s, env \text{ in } A[x_1]$
	$\to \text{let } \{x_i = x_{i+1}\}_{i=1}^{n-1}, x_n = \lambda y.s, env \text{ in } A[\lambda y.s]$
(sr,cp-e)	$\text{let } \{x_i = A_i[x_{i+1}]\}_{i=1}^{n-1}, x_n = A_n[y_1], \{y_j = y_{j+1}\}_{j=1}^{m-1}, y_m = \lambda z.s, env \text{ in } A[x_1]$
	$\to \text{let } \{x_i = A_i[x_{i+1}]\}_{i=1}^{n-1}, x_n = A_n[\lambda z.s], \{y_j = y_{j+1}\}_{j=1}^{m-1}, y_m = \lambda z.s, env \text{ in } A[x_1]$

Figure 2.1: Standard reduction rules from [2]

Often, there are laid out additional rules that are not necessarily required for reduction but can be used for optimization. We like to mention the garbage collection rules (gc1) and (gc2) from [2, 4], since they are proven to be sound and apply well to some expressions of this work. Both rules remove definitions from the environment if they are not relevant to the evaluation of the in-expression.

(gc1)	$\text{let } env_1, env_2 \text{ in } s \to \text{let } env_1 \text{ in } s$	if $LV(env_2) \cap (FV(env_1) \cup FV(s)) = \varnothing$
(gc2)	$\text{let } env \text{ in } s \to s$	if $LV(env) \cap FV(s) = \varnothing$

Figure 2.2: Garbage collection rules from [2, 4]

2.5 Variables

Variables of an environment that are on the left side of assignments are called *left variables* LV. More formal: $x \in LV$, iff $\exists s_i$ such that $x_i = s_i \in env$. In an expression, the variables can occur either free or bound. A variable is bound, if it is the binder of an abstraction, i.e. the variable x in $\lambda x.s$, or when it is a left side of an assignment in the environment, (let env in s) and $x = s_i \in env$. A variable is free, when it is not bound. For any expression, the set of bound variable $BV(e)$ and the set of free variables $FV(e)$ can be computed by the following rules:

$$\begin{aligned}
&BV(x) && = \varnothing\\
&BV(\lambda x.s) && = \{x\} \cup BV(s)\\
&BV(s\ t) && = BV(s) \cup BV(t)\\
&BV(\text{Let } env \text{ in } s) && = (BV(env) \cap FV(s)) \cup BV(s)\\
&BV(s \oplus t) && = BV(s) \cup BV(t)\\
&BV(env) && = \bigcup_{x_i=s_i} \{x_i\} \cup BV(s_i)\\
\\
&FV(x) && = x\\
&FV(\lambda x.s) && = FV(s) \setminus \{x\}\\
&FV(s\ t) && = FV(s) \cup FV(t)\\
&FV(\text{Let } env \text{ in } s) && = (FV(s) \cup FV(env)) \setminus BV(env)\\
&FV(s \oplus t) && = FV(s) \cup FV(t)\\
&FV(env) && = \bigcup_{x_i=s_i} FV(x_i) \setminus \bigcup_{x_i=s_i} x_i
\end{aligned}$$

Figure 2.3: Definition of free variables FV and bound variables BV.

These computation rules are the conventional ones and have only been extended for the cases of let and prob expressions. It should be stated, that the expression copied in a (sr,cp-in) or (sr,cp-e) reduction contains variable names that conflict with other variable names. The distinct variable convention (DVC) may be violated. In this case, a variable can have free and bound occurrences at the same time, so $FV(s) \cap BV(s) \neq$ or free variables can be captured, leading to an incorrect change of the expression's semantics. To be safe, we assume that the DVC is always satisfied, or perform an α-conversion implicitly.

2.5.1 α-conversion

In lambda calculus, two expressions are considered alpha equivalent if they differ only in the choice of variable names. Two expressions s and t are α-equivalent if and only if there exists a renaming that turns the expression into equal expressions by definition. The α-conversion defines the rule of renaming the variables of an expression without changing the meaning of a program. The distinct variable convention must be satisfied strictly. Otherwise, free variables may be captured by a renamed variable, changing the meaning of the expression. Every time a variable is renamed, a fresh variable name is chosen such that the DVC is satisfied. Later on, we take usage of alpha conversion for the standard reduction rules (sr,cp-in) and (sr,cp-e). These rules copy expressions from one into another. By that, the DVC might be violated.

2.5.2 Substitution

Substitution is the process of replacing all occurrences of a variable x by a variable x' in the expression s denoted by $s[x'/x]$. Sometimes, the notion of $s[x \to x']$ is used, like in [6]. This might be preferable, as it is self-explanatory which variable is the replaced and which is the replacing one.
Substitution is used to rename the variables during alpha conversion or when a definition is replacing a variable during reduction.

2.6 Weighted reduction

The evaluation using the standard reduction rules 2.1 is non-deterministic. Reducing the prob-operator yields in only one result that is either the sub-expression to the left or the sub-expression to the right. It is analogue to the run of a random experiment. Because the parameter p of the prob-operator is given, one can reason about the solution without performing the non-deterministic reductions (sr,probl) and (sr,probr). Therefor, the two sub-expressions on both sides are reduced at once and the probabilities are tracked. The new reduction rule ($\oplus_p$) is the union of (sr,probl) and (sr,probr). In addition, the expressions are provided with probabilities. This is called *weighted expression.*

Definition 1. A *weighted expression* is a pair (p, s) where $p \in [0, 1]$ is a probability and s an expression. If the notation does not lead to misunderstandings, the shorter notation of ps can be used interchangeably.

We supplement the reduction definition in such a way that the reduction result is provided as an weighted expression. For non-prob-reductions $s \xrightarrow{sr,a} t$ and $a \notin \{probl, probr\}$, the probability stays unchanged, for prob-reductions there are two weighted expressions. For collecting all possible results, probability distributions are used.

$$(\oplus_p) \quad R[s \oplus_p t] \to [pR[s], (1-p)R[t]]$$

Figure 2.4: Reduction rule $\oplus_p$

Now, the result for prob-expressions are not longer a single element rather than a distribution. The reduction step is now deterministic. To simulate the non-deterministic standard reduction in retrospect, a single expression can be sampled from the distribution by a function that uses a random generator afterward. The standard reduction rules have to be adopted to work on distributions as an input. Since distributions are covered in chapter 4, we're postponing this for now.

2.7 Program Calculus Λ_{PNeed}

Calculus Λ_{PNeed} is a simplification of the calculus Λ_{PNeedR}. It only differs in the definition of the prob operator that does not provide the parameter p and hence can not be biased. Instead, the probability of p is fixed to a constant value of $\frac{1}{2}$. Thus, one can abbreviate the notation to $s \oplus t$ by omitting the p. Since this can easily be simulated in $\Lambda_{PNeed} \subseteq \Lambda_{PNeedR}$ holds. The reduction rule must be adopted too, as shown in 2.5. Later, it is shown, that expressions from Λ_{PNeedR} can also be simulated in Λ_{PNeed}. The other direction is trivial.

$$(\oplus) \quad R[s \oplus t] \to [\tfrac{1}{2}R[s], \tfrac{1}{2}R[t]]$$

Figure 2.5: Reduction rule $\oplus$

3 Equivalence

The notion of equality must first be defined, since equality depends on different measures that strongly depends on the respective context. For example, the equality can be defined on the number of computational instructions during the evaluation process, even if this makes little sense in most of the cases. Then two programs would equal, if they share the same number of operations. In the scope of this thesis, we use *contextually equivalence* $\sim_C$ and *distribution equivalence* $\sim_D$ as the measure of all things. Before we delve into this, we give a brief overview of equivalences that are commonly used to elucidate the connections. Clarify the differences to contextual equivalence.

Semantic Equivalence

The semantics of a program is given by the evaluation strategy [1]. Accordingly, two programs are semantically equivalent if they are the same with regard to all evaluation strategies. [7] states, that it implies operational equivalence but not vise versa. In this thesis, we will only focus on the call-by-need strategy and will therefore not use semantic equivalence as a criterion for equality.

Operational Equivalence

Two terms are operationally equivalent if either can be removed from a program and replaced by the other without altering the behavior of the program. The behavioral equality coincide with denotational equality [7].

3.1 Contextual Equivalence

Contextually equivalence is weaker than operational and semantic equivalence, since it only compares the behavior of two different programs or expressions within a specific evaluation strategy that defines the semantics. Two expressions are contextually equivalent if and only if the expected convergence of the expressions plugged into any program context is always the same [2]. Thus, contextual equivalence implies operational equality. The other direction does not hold.

Definition 2 (Contextual Equivalence)**.** Let s and t be expressions. Let $s \leq_C t$ be the *contextual approximation* and its symmetry $s \sim_C t$ denote the *contextual equivalence.*

$$s \leq_C t \text{ iff } ExCv[C[s]] \leq ExCv[C[t]] \text{ with } p \leq q$$
$$s \sim_C t \text{ iff } s \leq_C t \wedge s \geq_C t$$

Here, $ExCv$ denotes the *expected convergence.* In [2] was shown, that this measure suffices to argue contextually equivalence.

3.2 Expected Convergence

Expected convergence of an expression s, denoted by $ExCv(s)$, is the probability of an expression being evaluated to a WHNF. For a deterministic program, there are two options: The evaluation either reduces to a WHNF in finite amount of reduction steps $ExCc(s) = 1$, or the reduction sequence is transfinite $ExCv(s) = 0$. In contrast, probabilistic programs can evaluate to multiple results. The evaluation is a distribution over multiple expressions. Now, the expected convergence is summed up for all the evaluations being in WHNF, weighted by their probability.

$$ExCv(s) = \textstyle\sum_{(p_i,s_i)\in Eval(s)} p_i$$

In the probabilistic setup, we replace observing termination with observing the expectation of termination i.e. the limit of the sum of the probabilities of all successful evaluations, where contextual equivalence holds, if this expected termination is the same for $C[s]$ and $C[t]$. It is known, that the observation of expected convergence is sufficient to state contextually equivalence. This holds because of the quantization of contexts, it is always possible to find a context that makes the differences between the expressions recognizable by modifying the probability of convergence [2, 3].

There is another equivalence, namely *distribution equivalence*, that is of interest in our context. Since distribution equivalence requires the knowledge of distributions and is defined within the next chapter 4.

4 Distributions

By extending the probabilistic operator $\oplus_p$, the result of a program is not a deterministic result but a probability distribution over all possible results. The equality of programs can now be argued by distributive equivalence. For example, [3] showed, that contextually equivalence implies distributive equivalence in a calculus called PCF, which is simply typed and uses Church encoding. If this applies to other languages too, like our untyped calculus with open expressions, is still an open question. In order to compare distributions, the distributions must be computed first. There are expressions for which the evaluation fails even though an intuitive solution is available. There exists several reasons for this. The first is founded in the halting problem. It is undecidable whether the evaluation of an expression will be caught in an infinite loop. Thus, the program must evaluate all results to be able to compare them. If only one of the results is transfinite, the evaluation will not terminate. But infinite loops may be a desired element of some programs. Besides that, some expressions can produce infinitely long distributions that can not be computed on real-world machines that have finite amount of storage, and others can be recursive such that the standard reduction can not evaluate them in finite time. The latter case may be a lack of transformation rules that do not take recursion into account. One goal of this thesis is the development of an evaluation strategy, that can evaluate expressions which are not reducible with standard reduction. First of all, we provide the definition of distributions, whereby we stay close to the notion of [4, 8].

Definition 3 (Distribution)**.** A *distribution* $dst(s)$ of an expression s is a unordered set $[(p_i, s_i)|i \in I]$ of weighted expressions (p_i, s_i) over the index set I where $p_i \in [0, 1]$ being the probability of s_i. The probability sum bounded by $\sum_{i \in I} p_i \leq 1$ and all s_i must be distinct.

Definition 4 (Multi-Distribution)**.** A *multi-distribution* $mdst(s)$ of an expression s is a finite, unordered set of tuples $[(p_i, s_i)|i \in I]$ of weighted expressions (p_i, s_i) over the index set I where $p_i \in [0, 1]$ being the probability of s_i. The probability sum bounded by $\sum_{i \in I} p_i \leq 1$. In $mdst$ an expression s_i can be contained multiple times.

Definition of $mdst$ is similar to the one of dst. The difference is, that in $mdst$ the s_i can occur more than once, e.g. every single evaluation that result contains s_i can have its own WEP in the $mdst$. A dst can be obtained from a $mdst$ by grouping all equivalent entries and summation of their probabilities. This does not work in practice, because it is not possible to decide which expressions are equivalent in general [2]. It follows that $dst \subseteq mdst$. The other direction is not guaranteed in practice.

4.1 Distribution Equivalence

Two expressions s and t are *distribution equivalent*, if their distributions are equal.

Definition 5 (Distribution Equivalence). Let s and t be expressions, and let $(p_i s_i) \in Eval(s)$ and $(q_j t_j) \in Eval(t)$ the weighted expression pairs of WHNFs in the evaluations. *Distribution approximation* $s \leq_D t$ and *distribution equivalence* $\sim_D$ hold like follows:

$$\begin{gathered} s \leq_D t \text{ iff } \forall s_i \, \exists t_j \text{ with } s_i \sim_C t_j \wedge p_i \leq t_j \\ s \sim_D t \text{ iff } s \leq_D t \wedge s \geq_D t \end{gathered}$$

This means that every expression in one of the distributions has an equivalent in the other distribution, and they have the same probability. This definition can also be extended to work on multi distributions. Then the probability sum of all contextual equivalent expressions must be equal in both distributions.
The way distribution equivalence is defined automatically implies contextual Equivalence. In other words, contextual equivalence of an expression can be shown by verify distributional equivalence. In order to compare two programs based on their distributions, the distributions must be able to be calculated. This is a challenging task and will be researched throughout the next sections.

4.2 Reduction on Distributions

In order to compute multi-distributions on expressions, the standard reduction has to be adopted to work on distributions. This can be done with the *lifting* $\Rightarrow$ like it is provided as rules set out in [4]:

$$\frac{}{[s] \Rightarrow [s]}\, L1 \qquad \frac{s \to m}{[s] \Rightarrow m}\, L2 \qquad \frac{\forall i \in I : [p_i s_i] \Rightarrow m_i}{[p_i s_i | i \in I] \Rightarrow \sum_{i \in I} p_i m_i}\, L3$$

This lifting lets the standard reduction apply to the elements of distributions. The output then is also a distribution. When the initial distribution only consists of a single element (p_i, s_i), two cases occur: an s_i is in WHNF, $L1$ could apply, and no standard reduction is performed. Otherwise, when s_i is reducible, $L2$ applies, when the initial distribution consists of many weighted expressions (p_i, s_i). The weighted reduction can be applied by isolating the weighted expressions from the distribution, performing a standard reduction afterwards, and reassemble the distribution by concatenating all the results. Since a distribution contains $|I|$ many expressions s_i, the same number of standard reductions can be performed every reduction step of $\Rightarrow$ at once theoretically. For the sake of simplicity, we don't want to distinguish between the lifting and standard reduction from now on and write $\to$ as the union of both reductions, assuming, that the right rules is chosen implicitly.

We call a distribution $dist(s)$ to be in $WHNF$ when all expressions s_i are in $WHNF$. Some of the expression s_i may divergent. A common practice is to drop them out of the distribution and add a placeholder $(q, \perp)$ with $q = 1 - \sum_{i \in I} p_i$. Since divergent expressions can neither be distinguished nor have any measurable properties, this does not result in any loss of data. $\perp$ is explained in Section 5.2 more detailed. The expected convergence of distributions than is the sum over all probabilities p_i where s_i is in $WHNF$.

$$ExCv[dst(s)] = ExCv([p_i s_i | i \in I]) = \sum_{s_i \in WHNF} p_i$$

The comparison $dst(s) = dst(t)$ is not applicable in practice because finding all equivalent values is undecidable [2]. This is easy to see using the following example from [1] that uses church encodings.

$$(Y\ (\lambda x.\lambda y.(y \oplus (x\ (succ\ y))))) \ 0$$

This expression applies two different operations with a probability of 0.5 each. Either it outputs the argument or it calls itself recursively with the argument incremented by one. This results in a distribution $\left[\frac{1}{2}1, \frac{1}{4}2, \frac{1}{8}3, ...\right]$ of transfinite length which can neither be compared nor computed using operational semantics as they inherently ascribe finiteness. For this reason we put these expressions aside and in this thesis concentrate on computable expressions and their distributions. This although implies that one can convert the $mdst$ to dst and hence we use "distribution" (dst) as a synonym for both from now on.

4.3 Distributions as Linear Equations

It is a challenging task to compute the distributions for an expression. We start simple by providing axiomatic statements, for which the distributional equivalence is obvious. Later on, we work out a bunch of transformation rules that operate on distributions. We will see that this workaround provides more freedom during the calculation of distributions. We begin with an axiom that states the distribution equivalence of an expression compared with its distribution and WEP.

$$dist(s) = [s] \tag{4.1}$$

$[s]$ is the short form of $[(1, s)]$ This must follow from the weighted reduction of length 0. With this, we can show, that the distributions can be denoted like linear equations. This is shown in figure 4.1, by applying the concatenation rule, the point-wise multiplication and the axiom 4.1 sequential. Later we will often use this form of notation when calculating with distributions become necessary, and the notation is quite practical in our opinion.

An environment becomes a system of linear equations.

$$env = [x_i = s_i | i \in I] = [dst(x_i) = dst(s_i) | i \in I]$$

In former research of [4], the point-wise multiplication $k \cdot dst$ with a scalar k and the concatenation $dst_1 + dst_2$ defined on distributions are described. We complement the arithmetic operators with division, that can be simulated by the point-wise multiplication

$$\begin{aligned} dst(s) &:= [p_i s_i | i \in I] \\ &= [p_1 s_1, ..., p_n s_n] \\ &= [p_1 s_1] + ... + [p_n s_n] \\ &= p_1 [s_1] + ... + p_n [s_n] \\ &= p_1 s_1 + ... + p_n s_n \end{aligned}$$

Figure 4.1: Derivation of distributions for linear equations

using the reciprocal of the scalar, subtraction by changing the signs and the cross product of two distributions that is the pairwise application as shown below.

$$\begin{aligned} \tfrac{dst}{k} &= \tfrac{1}{k} \cdot dst \\ dst_1 - dst_2 &= dst_1 + (-1) \cdot dst_2 \\ dst_1 \cdot dst2 &= \textstyle\sum_{(p_i, s_i) \in dst_1, (q_i, t_i) \in dst_2} p_i q_i (s_i\ t_i) \end{aligned}$$

Figure 4.2: Arithmetic operators on distributions

At this point, arithmetic operators and precedence rules can be applied as usual. One may wonder about the negative probabilities that may appear. This contradicts with the norms of probabilities, but is not harmful when the overall result meets the requirements.

4.4 Distribution Decomposition

In some cases, a definition of an expression can be decomposed into a sum of distributions for sub-expressions. We introduce a set of distribution decomposition rules you can find in figure 4.3 that can be applied without performing reductions.
The last expression enables many is by far the trickiest and is developed in the next chapters. These rules do not reduce the expressions, but act like a backtracking of probabilities over the tree of evaluation paths. Poorly, the backtracking stops whenever an abstraction is in scope, since abstractions only resolves, when they are needed. A rule for abstractions was considered, but turned out to be wrong soon. The inequality can be shown, providing a counterexample from [2].

$$dst(\lambda x.s) \neq [p_1 \lambda x.dst(s_1), ..., p_n \lambda x.dst(s_n)]$$

$$\begin{array}{lcl}
dst(x) & = & [x] \\
dst(\lambda x.s) & = & [\lambda x.s] \\
dst(s\ t) & = & dst(s) \cdot dst(t) \\
dst(s \oplus_p t) & = & p \cdot dst(s) + (1-p) \cdot dst(t) \\
dst(\text{let } env \text{ in } s) & = & [\ \text{let } env \text{ in } s'\ ] \\
 & & \text{where } s' = s[x_i \mapsto dst(s_i)]\ \forall i \in I
\end{array}$$

Figure 4.3: Distribution Decomposition Rules

Assume the rewrite rule is correct, both sides are equal and let s_1 and s_2 be two expressions.

$$\begin{array}{ll}
s_1 := \text{let } f = \lambda x.\lambda y.x \oplus \lambda x.\lambda y.y & \text{in } f\ (f\ x_1\ x_2)(f\ x_3\ x_4) \\
s_2 := \text{let } f = \lambda x.\lambda y.x \oplus y & \text{in } f\ (f\ x_1\ x_2)(f x_3\ x_4)
\end{array}$$

Since the distributions of both expressions would calculate to $f = \left[\frac{1}{2}\lambda x.\lambda y.x, \frac{1}{2}\lambda x.\lambda y.y\right]$, both expressions assume equal. But performing standard reduction on both expressions leads to $e_1 = \left[\frac{1}{2}x_1, \frac{1}{2}x_2\right]$ and $e_2 = \left[\frac{1}{4}x_1, \frac{1}{4}x_2, \frac{1}{4}x_3, \frac{1}{4}x_4\right]$. Since both distributions are different, the assumption is violated.
The differences can be explained by the order of evaluation and is in fact an issue of the call-by need-evaluation. In e_1 the function is evaluated first. The result of f is then copied three times into the in expression. The standard reduction rules (sr,cp-in) and (sr,cp-e) can perform copy operations for common. Therefore, every copy is the same. In contrast, the definition of f in e_2 wraps the prob operator into the abstractions, so the expression already exists as a WHNF. It is copied three times before it can be evaluated. Hence, every occurrence of f is evaluated independently and can lead to different results. A real-world analogy is the difference between rolling two dice in contrast to multiplying a dice roll by 2. As a conclusion to the previous observation, abstractions must be maintained as they occur in the environment. Nevertheless, the distributive rules can be applied to the definitions recursively.

5 Recursion

There exists certain categories of λ-terms that can not be evaluated by the previous introduced techniques. One set that hold this property is explained in [9] namely the set Λ^∞ of infinite λ-terms. These terms may be transfinite in the length of input, which in turn means, that the sequence of reduction must be trans-finite as well. In this section we will not discuss the properties of those, since these are quite impractical and the discussion would be beyond the scope of the thesis. We will notice that the class of recursively defined λ-terms is also infinite with respect to standard reduction, but offers more possibilities in the evaluation process.

The operational semantics discussed above are limited because evaluation fails for many expressions that even have a valid meaning. Mainly responsible are infinite reduction sequences that can arise through recursion. In the introduction to this chapter, certain recursive examples are given for which the evaluation fail.

5.1 Ω

The expression $\Omega = \lambda x.(x\ x)\ \lambda x.(x\ x)$ is a prominent function that does not evaluate. Nevertheless, we show the first reduction steps using our introduced operational semantics to clarify the problem.

$$
\begin{array}{ll}
 & \Omega \\
= & (\lambda x_2.(x_2\ x_2))\ (\lambda x_1.(x_1\ x_1)) \\
\xrightarrow{sr,lbeta} & \text{let } x_2 = \lambda x_1.(x_1\ x_1) \text{ in } (x_2\ x_2) \\
\xrightarrow{sr,cp-in} & \text{let } x_2 = \lambda x_1.(x_1\ x_1) \text{ in } ((\lambda x_3.(x_3\ x_3))\ x_2) \\
\xrightarrow{sr,lbeta} & \text{let } x_2 = \lambda x_1.(x_1\ x_1) \text{ in let } x_3 = x_2 \text{ in } (x_3\ x_3) \\
\xrightarrow{sr,llet-in} & \text{let } x_2 = \lambda x_1.(x_1\ x_1), x_3 = x_2 \text{ in } (x_3\ x_3) \\
\xrightarrow{sr,cp-in} & \text{let } x_2 = \lambda x_1.(x_1\ x_1), x_1 = x_2 \text{ in } ((\lambda x_4.(x_4\ x_4))\ x_3) \\
 & \dots \\
\xrightarrow{sr,llet-in} & \text{let } x_1 = \lambda x_2.(x_2\ x_2), x_2 = x_1, \dots, x_{n-1} = x_n \text{ in } (x_2\ x_2) \\
 & \dots
\end{array}
$$

It is easy to see, that the reduction runs into a periodic behavior. In every three reduction steps, the expressions are alpha-equivalent repetitively and an assignment $x_i - 1 = x_i$ is added that exactly reflects the renaming of the α-conversion. Note that the alpha-equivalence yields from the renaming we perform after every (sr,cp-in) reduction to guarantee the DVC.

5.2 Black Hole $\perp$

Another prominent expression is the *black hole* $\perp$.

$$\perp = \text{let } x = x \text{ in } x$$

This expression embodies non-termination and is used for replacing other expressions that share the property of non-termination. Non-terminating expressions are interchangeable without changing the meaning of a program, since the property of non-termination is the only observable in this type of expression. For this reason, other non-terminating expressions are often replaced by the black hole, since it offers the simplest implementation of non-termination. When trying to reduce this expression using standard reduction, the variable x of the in-expression is in the scope of the reduction context. The (sr,cp) rules may apply. Thus, the reduction tries to find the expression at the end of a variable chain or context chain. This will fail for the black hole, because the chain $\{x = x\}_{i=1}^{\infty}$ is of infinite length. With the rules of distributions stated so far, recursion can be removed in some cases without the need for reduction. Linear equation can be extracted out of a definition with the rules of figure 4.3. Applying this on $\perp$, we get the following result:

$$\begin{aligned} & dist(x) = dist(x) & \\ = \quad & x = x & \mid -x \\ = \quad & 0 = 0 & \end{aligned}$$

The linear equation has no solution, as is to be expected for a divergent expression like $\perp$.

5.3 Y Combinator

The *fixpoint combinator* Y is a well known function. It is often used to implement recursion in calculi, that does not offer recursion native. The Y Combinator implements self application of a function f.

$$Y = \lambda f.\lambda x.f\ (x\ x)\ \lambda f.\lambda x.f\ (x\ x)$$

$Y\ f$ results in the infinite sequence of self application $f\ (f\ (f\ (...(Yf)...)))$. The operational semantics again fails to reduce those expressions. But depending on the function f, there can be expressions that are logically solvable, like shown below.

Example 1.
$$\begin{aligned} & Y\ \lambda x.(x \oplus t) \\ \sim_d \quad & \text{let } x = x \oplus t \text{ in } x \end{aligned}$$

This expression points out to be probabilistic convergent and converges to t with probability 1. But this function contains recursion and hence can not be evaluated using the operational semantic used so far.

$$
\begin{array}{ll}
 & Y\ \lambda x.(x \oplus t) \\
= & (\lambda f.(\lambda x_2.f\ (x_2\ x_2))\ (\lambda x_1.f\ (x_1\ x_1)))\ \lambda x.(x \oplus t) \\
\xrightarrow{sr,lbeta} & \text{let } f = \lambda x.(x \oplus t) \text{ in } (\lambda x_2.f\ (x_2\ x_2))\ (\lambda x_1.f\ (x_1\ x_1)) \\
\xrightarrow{sr,lbeta} & \text{let } f = \lambda x.(x \oplus t) \text{ in let } x_2 = \lambda x_1.f\ (x_1\ x_1) \text{ in } f\ (x_2\ x_2) \\
\xrightarrow{sr,llet-in} & \text{let } f = \lambda x.(x \oplus t), x_2 = \lambda x_1.f\ (x_1\ x_1) \text{ in } f\ (x_2\ x_2) \\
\xrightarrow{sr,cp-in} & \text{let } f = \lambda x.(x \oplus t), x_2 = \lambda x_1.f\ (x_1\ x_1) \text{ in } (\lambda x_3.x_3 \oplus t\ (x_2\ x_2) \\
\xrightarrow{sr,lbeta} & \text{let } f = \lambda x.(x \oplus t), x_2 = \lambda x_1.f\ (x_1\ x_1) \text{ in let } x_3 = (x_2\ x_2) \text{ in } (x_3 \oplus t) \\
\xrightarrow{sr,llet-in} & \text{let } f = \lambda x.(x \oplus t), x_2 = \lambda x_1.f\ (x_1\ x_1), x_3 = (x_2\ x_2) \text{ in } (x_3 \oplus t) \\
\xrightarrow{sr,prob} & \big[\frac{1}{2} \text{ let } f = \lambda x.(x \oplus t), x_2 = \lambda x_1.f\ (x_1\ x_1), x_3 = (x_2\ x_2) \text{ in } x_3, \frac{1}{2}t \\
\xrightarrow{sr,cp-in} & \big[\frac{1}{2} \text{ let } f = \lambda x.(x \oplus t), x_2 = \lambda x_1.f\ (x_1\ x_1), x_3 = (x_2\ x_2) \text{ in } (x_2\ x_2), \frac{1}{2}t \\
\xrightarrow{sr,cp-in} & \big[\frac{1}{2} \text{ let } f = \lambda x.(x \oplus t), x_2 = \lambda x_1.f\ (x_1\ x_1), x_3 = (x_2\ x_2) \text{ in } (\lambda x_4.f\ (x_4\ x_4)\ x_2), \frac{1}{2}t \\
\xrightarrow{sr,lbeta} & \big[\frac{1}{2} \text{ let } f = \lambda x.(x \oplus t), x_2 = \lambda x_1.f\ (x_1\ x_1), x_3 = (x_2\ x_2) \text{ in let } x_4 = x_2 \text{ in } f\ (x_4\ x_4), \frac{1}{2}t \\
\xrightarrow{sr,llet-in} & \big[\frac{1}{2} \text{ let } f = \lambda x.(x \oplus t), x_2 = \lambda x_1.f\ (x_1\ x_1), x_3 = (x_2\ x_2), x_4 = x_2 \text{ in } f\ (x_4\ x_4), \frac{1}{2}t \\
 & \dots
\end{array}
$$

We can observe similarities to the reduction of Ω, but the periodic appearance of terms can only be found in sub-expressions. We explicitly mention the omission of some of the definitions in the environment that belongs to the expression t for better readability. This is sound due to the garbage collection rules (gc1) and (gc2) one can find in [2, 4]. The distribution is infinite and therefore can not evaluate fully. However, approximations can be computed, like the Monte Carlo strategy of [4] does. This observation has enormous impact for the comparability of expressions, since only accurately measured distributions can be checked for equality. However, this compromise, which allows for the inaccuracies of approximations, is not in the spirit of this work. A more accurate analysis of the $Y\ \lambda x.(x \oplus t)$ expression can take place in the mathematical sense by structural analysis of repetitive pattern. In our example, we apply geometric series. The resulting expression then will converge to t with probability 1.

$$\begin{array}{ll}
\sim_D & [\tfrac{1}{2}\ Y\ \lambda x.(x \oplus t,\ \tfrac{1}{2}t] \\
\sim_D & [\tfrac{1}{2}\ [\tfrac{1}{2}\ [\tfrac{1}{2}\ [...],\ \tfrac{1}{2}k],\ \tfrac{1}{2}t],\ \tfrac{1}{2}t] \\
= & [(\tfrac{1}{2} + \tfrac{1}{4} + \tfrac{1}{8} + ...)t] \\
= & t
\end{array}$$

The accurate distribution can now be used to check distribution equality with other programs. Deriving this solution by hand is not particularly difficult and can be solved by just apply geometric rows to the prefactor of t. It is even obvious that the programs Y $\lambda x.(x \oplus t)$ and t are equal in a semantic manner, which becomes clear with the following analogy: Flip a coin. If head is up, repeat the flip (represented by $\lambda x.x$), otherwise stop. At some point, "number" (represented by t) has to be up by arbitrary high probability, since it converges to 1 in the limit of the flips. However, it is very frustrating to formulate rules, or even to implement program code for solving strategy like presented. This is due to the quantity of possible functions, which requires individual solving methods. The question arises, whether a solution can always be computed. We know, the expression Ω is not even computable. There is a large set of expression, that behave similar. It turns out that this is again related to the halting problem and thus not decidable in general. Nevertheless, in chapter 6 and 7 is shown, that for some expressions like the last example, a solution even can be computed automatically.

5.4 Direct recursion

Again assume example 1. The evaluation using standard reduction has a transfinite reduction sequence. By decomposition of the distributions the resulting equation system can be solved.

$$\begin{array}{ll}
 & x = x \oplus t \text{ in } x \\
\sim_D & x = dst(x \oplus t) \\
\sim_D & x = \tfrac{1}{2}dst(x) + \tfrac{1}{2}dst(t) \\
\sim_D & x = [\tfrac{1}{2}x] + [\tfrac{1}{2}t] \\
\sim_D & x = \tfrac{1}{2}x + \tfrac{1}{2}t
\end{array}$$

Then the definition of x can be simplified, since its definition contains itself.

$$\begin{array}{lll} & x = \frac{1}{2}x + \frac{1}{2}t & | - \frac{1}{2}x \\ \sim_D & \frac{1}{2}x = \frac{1}{2}t & | \cdot 2 \\ \sim_D & x = t & \end{array}$$

After that, the definition $x = t$ will replace the old expression and the recursion is solved. In chapter 8 we show, how the distribution can be converted back into an expression of λ_{PNeedR}. We conclude the observations of this direct form of recursion by stating a general rule.

Direct recursion can be found in environments, where the distribution of a definition contains its left variable itself.

$$x_1 = [p_1 x_1, p_2 x_2, ..., p_n x_n] \Leftrightarrow x_1 = \begin{cases} \frac{1}{1-p_1}[p_2 x_2, ..., p_n x_n], & \text{for } 0 \leq x < 1 \\ \perp, & \text{for } p_1 = 1 \end{cases} \tag{5.1}$$

If the probability of self occurrence is 1, the equation is a black hole. If the probability of the self occurrence is smaller than 1, the variable can be removed from the distribution. To maintain the value of expected convergence, the probabilities of the remaining expressions are multiplied by a scalar. Proof: This follows by simply applying arithmetic operators.

$$\begin{array}{rcll} x_1 & = & [p_1 x_1, p_2 x_2, ..., p_n x_n] & | - [p_1 x_1] \\ (1 - p_1)\, x_1 & = & [p_1 x_1, p_2 x_2, ..., p_n x_n] & | \frac{1}{1-p_1} \\ x_1 & = & [\frac{p_2}{1-p_1} x_2, ..., \frac{p_n}{1-p_1} x_n] & \end{array}$$

□

5.5 Dependent Graph

The definitions in an environment may contain variables that are dependent to other definitions, i.e. the definition contains other left variables. For example, the standard reduction rule (sr,cp-in) use those to traverse through the definition to find the end of a chain of dependencies. We distinguish two kinds of chains: *variable chains* $\{x_i = x_{i+1}\}_{i=1}^{n-1}$, where the expressions of the definitions are given by a single binder variable. *context chains* $x_i = \mathbb{A}_i[x_{i+1}]_{i=1}^{n-1}$, where the left variables are in the scope of the application contexts $\mathbb{A}_i$. These are part of (sr,cp-e) and (sr-llet-e). These chains represent sequences of variables. However, in the probabilistic setup, these chains can be forked due to the prob-operator with different probabilities, or it can be joined by invoking a common variable from multiple definitions. We want to emphasize again that the prob-operator produces a probability distribution over the dependencies. We define a dependency as follows:

Definition 6 (Dependency of Environmental Variables)**.** We say a binder variable x_i to be *dependent* of a binder variable x_j s, iff $(p, s_j) \in dst(x_i)$ for all arbitrary p.

Again, computing $dst(s)$ of an expressions s is undecidable in general. This means, that not all dependencies can be recognized. Invoking the distribution decomposition rules 4.3 is a compromise that is used in the interpreter implemented besides this thesis. Since the dependencies are present in an n-to-n relationship, a directed graph or forest called *Dependent Graph* can be used to represent them.
The dependencies stated so far can be visualized using directed graphs. The prob-operator forks the path of reduction. For the definitions in an environment, this means that a definition can depend on several left variables LV. Then, s_i is dependent to all left variables that occur in $dst(s_i)$ as an individual. To bring the distributions into a form, that the left variables occur as single elements, the distribution decomposition rules 4.3 can be applied. Let X be the set of all left variables. Let $S_i = dst(s_i)$ be the decomposition of the environmental expression s_i. Then, the set of nodes V of the dependent graph are given by the union of X and all environmental decomposition S_i.

$$\begin{aligned}
&S_i = dst(s_i)\\
&X = \textstyle\bigcup_{i\in I} x_i\\
&V = X \cup \textstyle\bigcup_{i\in I} S_i\\
&T = V \setminus X
\end{aligned}$$

We call a node $t \in T$ a *terminal*, if it is a node, that is not a left variable, since it has no definition and hence no dependency to other LVs. An edge $e_{u,v} = (u, v)$ with $u, v \in V$

exists if and only if v is contained in the definition of u and hence $u \in X$ follows. An edge weight $w(e_{i,j})$ is the probability of u evaluates to v. Notice that the sum of all outgoing edges is limited to 1. In our setting, a potential node $P(x)$ then is the overall probability distribution that one can obtain for the variable x. The resulting graph may be connected or can be decomposed into n components $C = \{C1, ..., C_n\}$. In the latter case, we only require the component C_i that is connected to x when computing $P(x)$.

Example 2. let $x_1 = t_1 \oplus x_2, x_2 = t_2 \oplus x_1$ in x_1

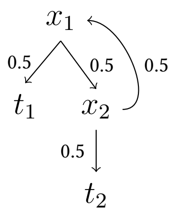

Figure 5.1: Dependent graph of example 2

The left variable x_1 of example 2 is dependent to x_2 and connected with an edge weight of $\frac{1}{2}$, since the expression pair $(\frac{1}{2}, x_2)$ occurs in the distribution of x_1.

$$dst(x_1) = dst(t_1 \oplus x_2) = \big[(\tfrac{1}{2}, t_1), (\tfrac{1}{2}, x_2)\big].$$

5.6 Indirect Recursion

Applying the direct recursive rule 5.1 would not suffice to remove recursion in general. For indirect recursive examples like the example 3, the left variables do not occur in the distribution of their distribution.

Example 3. let $f = a \oplus g, g = f \oplus b$ in f

This time, the environment has multiple left variables. In general, the distributions behave like homogeneous systems of linear equations. By substituting g into the definition of f the environment becomes direct recursive. Using the rule 5.1 solves the system of equations.

$$\begin{tikzcd}
 & f \arrow[dl] \arrow[dr] & & & g \arrow[dl] \arrow[dr] & \\
a & & g \arrow[urr, dashed] & f \arrow[ull, dashed] & & b
\end{tikzcd}$$

let $x_1 = t_1 \oplus x_2, x_2 = t_2 \oplus x_1$ in x_1

Figure 5.2: Indirect recursive environment

$$\begin{tikzcd}
 & f \arrow[dl] \arrow[dr] & & \\
a & & g \arrow[dl] \arrow[dr] & \\
 & f \arrow[uu, dashed, bend left] & & b
\end{tikzcd}$$

let $f = a \oplus (f \oplus b)$ in f

Figure 5.3: Direct recursive environment

$$\begin{aligned}
& f = dst(a \oplus (f \oplus b)) \\
& f = [\tfrac{1}{2}a, \tfrac{1}{2}(f \oplus b)] \\
& f = [\tfrac{1}{2}a, \tfrac{1}{4}f, \tfrac{1}{4}b] \qquad\qquad |- \tfrac{1}{4}f \\
\tfrac{3}{4} \quad & f = [\tfrac{1}{2}a, \tfrac{1}{4}b] \\
& f = [\tfrac{2}{3}a, \tfrac{1}{3}b]
\end{aligned}$$

For some expressions, the existence of mutual dependencies of the definitions makes it impossible to eliminate the indirect recursive function calls of f and g, since they will always appear alternately when they are substituted into each other.

Example 4. let $x_1 = x_2 \oplus x_3, x_2 = t_1 \oplus t_3, x_3 = x_2 \oplus t_2$ in x_1

$$\begin{tikzcd}
 & x_1 \arrow[dl, "\frac{1}{2}"'] \arrow[dr, "\frac{1}{2}"] & \\
x_2 \arrow[rr, leftrightarrow, "\frac{1}{2}"'] \arrow[d, "\frac{1}{2}"'] & & x_3 \arrow[d, "\frac{1}{2}"] \\
t_1 & & t_2
\end{tikzcd}$$

Figure 5.4: dependent graph of example 4

In example 4, the variables x_2 and x_3 are dependent on each other. Substituting the appropriate equations into the definition of x_1 always shows the other reciprocal variable. Thus, a technique for solving equation systems is required. Such strategies are discussed in the chapters 6 and 7.

6 Markov Chain

A *Markov chain* is a stochastic model to simulate state transitions that occur with certain properties. The state transition is given by a transition matrix P, i.e. a stochastic matrix, with the row-sum of 1. By multiplying the matrix on an initial state vector, a transition step is simulated. Multiplying the k-power of the matrix to the initial state simulates k steps at once. The limit of transition steps is of interest, as it includes the potential $P()$ of a left variable.

6.1 Transition Matrix

To get an efficient notation for the algorithmic level, a transition matrix P can be extracted from the environment. P is similar to the matrix one would construct from the dependency graph, but with small changes. This matrix is a stochastic matrix for which the row sum equals 1. To ensure this, self-loops may be added to the nodes. First, all the variables in the environment are collected and indexed in a vector $\vec{v} = v_1, ..., v_(n + m)$. Note that the order of the variables can be arbitrary, since the set of assignments is not ordered. But be aware, once the order of variables are chosen, they must be maintained, because it will associate the row of the matrix with the expressions of V. Remind, that X is the set of all left variables x_i with $1 \leq i \leq n$ and T be the set of all terminals t_j with $1 \leq j \leq m$. For simplification, we set $\vec{v}^T = (x_1, ..., x_n, t_1, ..., t_m)$ such that the binder variables are leading the terminals. The procedure takes the environment env as an input and computes the transition matrix like described from the following imperative pseudo algorithm:
P is first assigned by the $(n + m) \times (n + m)$ identity matrix. Thus, all row sums are equal to 1. Whenever a dependency is found for a left variable, the property of a transition is subtracted from the self-loop and added to the corresponding column. In this way, the terminals also receive a transition with the sum 1 in the form of a self-loop. This can be notified in the visualization as a graph where, in contrast to the dependent graph, the

```
transitionMatrix(env):
    //initialize P
    P = identityMatrix(n+m,n+m);

    //populate matrix P
    forall (x_i = s_i) in env:
        forall (p_jv_j) in dist(s_i)
            P_{i,i} -= p_j
            P_{i,j} += p_j
    return P
```

Figure 6.1: Imperative pseudoalgorithm for computing the transition matrix

transition graph strictly adheres to the property that the outgoing edges have a common edge weight of 1.
Revisit the expression from example 2. The difference between the dependent graph **??** and the transition graph **??** can be found in the self-loops. Using the expression of example 2 again, the transition graph adds self-loops to the terminals.

$$\begin{tikzcd}
& x_1 \arrow[dl, "\frac{1}{2}"'] \arrow[dr, "\frac{1}{2}"] & \\
t_1 \arrow[loop left, "1"] & & x_2 \arrow[ul, bend right, "\frac{1}{2}"'] \arrow[d, "\frac{1}{2}"'] \\
& & t_2 \arrow[loop left, "1"]
\end{tikzcd}$$

$$\text{let } x_1 = t_1 \oplus x_2, x_2 = t_2 \oplus x_1 \text{ in } x_1$$

Figure 6.2: transition graph of expression 2

let $x_1 = t_1 \oplus x_2, x_2 = t_2 \oplus x_1$ in x_1 from example **??**. the probability distributions are calculated first using the distribution rules.

$$\text{let } x_1 = [\tfrac{1}{2}t_1, \tfrac{1}{2}x_2], x_2 = [\tfrac{1}{2}t_2, \tfrac{1}{2}x_1] \text{ in } [x_1]$$

Let vector $\vec{v}$ be chosen as follows:

$$\vec{v} = \begin{bmatrix} x_1 & x_2 & t_1 & t_2 \end{bmatrix}$$

Then the respective transition matrix is shown in **??**

$$P = \begin{bmatrix} 0 & 0.5 & 0.5 & 0 \\ 0.5 & 0 & 0 & 0.5 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 6.3: Transition matrix of example 2

The transition matrix can now be used to calculate the probabilities of the variables and terminals. In this first approach, a state-based simulation is applied, which is implemented using Markov chains. The stochastic matrix P required and can be assigned by the transition matrix we have computed so far. Additionally, the vector p_0 being the initial state is required. To calculate the distribution of $P(x_i)$ the initial vector p^0 is set to $\vec{0}$ except for position i that is set to 1. Note that v can be omitted since it occurs on both sides, but this again clarifies, that the variables belong to the rows. When the initial vector is applied to the stochastic matrix, a state transition occurs. The number of state transitions performed can be determined by the power n of the matrix P.

$$I_i P^n v = p_n v \tag{6.1}$$

$p_n v$ is the probability distribution of the potential $P(x_i)$ if I_i is the i-th row of the $|v| \times |v|$ identity matrix. Thereby, a row of the identity matrix embodies the initial state of the transition experiment. The matrix multiplications simulate the flow of evaluation paths going through a chain of n different assignments. The probabilities of the variables to the end of the chains were approximated. Computing the limit of n is a very hard problem, and usually it gets approximated since it does not exist in general. Sadly, Markov Chains are not suitable for the comparison of distributions, since the imprecision does not remove recursive variables fully. Nevertheless, the method could be still beneficial for approximation, because the result is very precise even for small n and offers a runtime advantage in contrast to other approximation methods, e.g. the Monte Carlo method used in [4], which evaluates the expression with standard reduction a thousand times for precise results. One solution is, to hide those weighted expressions whose probability falls below a precision threshold. This method is tested in 9.

Example 5. let $x_1 = t_1 \oplus x_2, x_2 = t_2 \oplus x_1$ in x_1

$$P = \begin{bmatrix} 0 & 0.5 & 0.5 & 0 \\ 0.5 & 0 & 0 & 0.5 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathrm{p}_0 = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}$$

$$\mathrm{p}_1 = \begin{bmatrix} 0 & 0.5 & 0.5 & 0 \end{bmatrix}$$

$$\mathrm{p}_2 = \begin{bmatrix} 0.25 & 0 & 0.5 & 0.25 \end{bmatrix}$$

$$\mathrm{p}_3 = \begin{bmatrix} 0 & 0.125 & 0.625 & 0.25 \end{bmatrix}$$

$$\mathrm{p}_{10} = \begin{bmatrix} 0.0009765625 & 0 & 0.666015625 & 0.3330078125 \end{bmatrix}$$

The environment can be reassembled when the vectors are substituted into the equation 6.1. It is multiplied by v which contains expressions and thus the resulting vector is a distribution. We can observe that the values converge rapidly when increasing n. Sadly, some of the recursive variables are not vanished fully, and the method does not gain any advantage. An interesting observation made in 9, that these expressions are not even an approximation, but an equivalent representation of the expression. In the next section, we'll compute the exact result for this example and see, that this is not far from what this process produced.

7 Gaussian Elimination

The next approach uses Gaussian elimination to calculate the probability distribution of an environment exactly. This time, the transition matrix M is slightly modulated. In order to calculate the probability distribution $P(x)$, the method aims to solve the equation system given by the transition matrix for variable x. As an introduction to this method, we show how it can be calculated by hand. We then show the application of Gauss elimination, which has established itself as an algorithmic method.

By repeatedly substituting the equations into each other, we head for a result of the form $x = p_i t_i$, where $i \in 1, ..., n$ and t are sub-expressions or terminals not contained in the environment variables X. This equation finally represents the distribution. The main advantage of this method is, that when a solution exists, it is the precise distribution that can be needed to verify distribution equivalence.

Example 6. From the transition matrix of example 5 we can read off the linear equations.

$$\begin{cases} x_1 = \frac{1}{2}t_1 + \frac{1}{2}x_2 \\ t_1 = t_1 \\ x_2 = \frac{1}{2}t_2 + \frac{1}{2}x_3 \\ t_2 = t_2 \\ x_3 = x_1 \end{cases}$$

Note that in this term the constant equations are useless and can be omitted. We begin with the equation of x_1 since it is the entry point. Then we try to substitute the x_i until no x_i remains in the equation's right side of x_1. In this example, we first substitute x_2 and then x_3. This yields to the equation.

$$\begin{aligned}
&x_1 = \tfrac{1}{2}t_1 + \tfrac{1}{2}x_2 \\
&x_1 = \tfrac{1}{2}t_1 + \tfrac{1}{2}(\tfrac{1}{2}t_2 + \tfrac{1}{2}x_3) \\
&x_1 = \tfrac{1}{2}t_1 + \tfrac{1}{4}t_2 + \tfrac{1}{4}x_1 &&\mid -\tfrac{1}{4}x_1 \\
\tfrac{3}{4}\quad &x_1 = \tfrac{1}{2}t_1 + \tfrac{1}{4}t_2 + \tfrac{1}{4}x_1 &&\mid \cdot \tfrac{4}{3} \\
&x_1 = \tfrac{2}{3}t_1 + \tfrac{1}{3}t_2
\end{aligned}$$

The equation then only contains the t_i that can not reduce any further. We have just calculated the resulting distribution accurately with a small amount of effort. We can compare this distribution with the approximate result of example 5 that uses Markov chains, and find out, that the deviations are very small. But there are expressions were substituting expressions in like shown before causeds problems. If the equation contains multiple variables that are dependent on each other, the variables cannot be eliminated. In this situation, it can help to first reduce the equations of the problematic variables and then substitute the results into the equation of the called variable. Sometimes this must be done at several levels in a hierarchy.

Example 7. let $x_1 = x_2 \oplus x_3, x_2 = t_1 \oplus x_3, x_3 = x_2 \oplus t_2$ in x_1

$$\begin{tikzcd}
 & x_1 \arrow[dl, "\frac{1}{2}"'] \arrow[dr, "\frac{1}{2}"] & \\
x_2 \arrow[rr, leftrightarrow, "\frac{1}{2}"'] \arrow[d, "\frac{1}{2}"'] & & x_3 \arrow[d, "\frac{1}{2}"] \\
t_1 & & t_2
\end{tikzcd}$$

Figure 7.1: Auxiliary graph G_p

$$\begin{cases}
x_1 = \frac{1}{2}x_2 + \frac{1}{2}x_3 \\
x_2 = \frac{1}{2}t_1 + \frac{1}{2}x_4 \\
x_3 = \frac{1}{2}x_5 + \frac{1}{2}t_2 \\
x_4 = x_3 \\
t_5 = x_2
\end{cases}$$

In this example, we encounter a problem. The variables x_2 and x_3 are cyclic dependent to each other. This means that substituting one of the variables always produces the other variable. The elimination of both variables at once is not possible. But with a trick, the equation can still be solved. The distributions of x_2 and x_3 can be calculated first and then inserted into the equation of x_1.

$$
\begin{aligned}
x_2 &= \tfrac{1}{2}t_1 + \tfrac{1}{2}x_3 \\
x_2 &= \tfrac{1}{2}t_1 + \tfrac{1}{2}(\tfrac{1}{2}x_2 + \tfrac{1}{2}t_2) \\
x_2 &= \tfrac{1}{2}t_1 + \tfrac{1}{4}x_2 + \tfrac{1}{4}t_2 && | - \tfrac{1}{4}x_2 \\
\tfrac{3}{4} \quad x_2 &= \tfrac{1}{2}t_1 + \tfrac{1}{4}t_2 && | \cdot \tfrac{4}{3} \\
x_2 &= \tfrac{2}{3}t_1 + \tfrac{1}{3}t_2
\end{aligned}
$$

$$
\begin{aligned}
x_3 &= \tfrac{1}{2}x_2 + \tfrac{1}{2}t_2 \\
x_3 &= \tfrac{1}{2}(\tfrac{1}{2}t_1 + \tfrac{1}{2}x_2) + \tfrac{1}{2}t_2 \\
x_3 &= \tfrac{1}{4}t_1 + \tfrac{1}{4}x_3 + \tfrac{1}{2}t_2 && | - \tfrac{1}{4}x_3 \\
\tfrac{3}{4} \quad x_3 &= \tfrac{1}{4}t_1 + \tfrac{1}{2}t_2 && | \cdot \tfrac{4}{3} \\
x_3 &= \tfrac{1}{3}t_1 + \tfrac{2}{3}t_2
\end{aligned}
$$

These transformed equations no longer depend on variables and can therefore be successfully used in the definition of x_1.

$$
\begin{aligned}
x_1 &= \tfrac{1}{2}(\tfrac{2}{3}t_1 + \tfrac{1}{3}t_2) + \tfrac{1}{2}(\tfrac{1}{3}t_1 + \tfrac{2}{3}t_2) \\
x_1 &= \tfrac{1}{3}t_1 + \tfrac{1}{6}t_2 + \tfrac{1}{6}t_1 + \tfrac{1}{3}t_2 \\
x_1 &= \tfrac{1}{2}t_1 + \tfrac{1}{2}t_2
\end{aligned}
$$

In order to raise the process to an algorithmic level, the matrix notation is used again. The size of the matrix is $|X| \times |V|$. Every row is extracted from a definition within the environment in for of linear equations. To process all computation steps in one matrix, every $dst(s_i)$ is subtracted from x_i to bring it to the side of x_i. Then the dependencies occur as negative numbers. The matrix can be split into two separate matrices M_X, that is the $n \times n$ coefficient matrix of left variables $\vec{x} = (x_1, ..., x_n)^T$ and let M_T be the $n \times m$ coefficient matrix of the terminals $\vec{t} = (t_1, ..., t_m)^T$.

$$M_X\vec{x} = M_T\vec{t} \tag{7.1}$$

$$I\vec{x} = M_T'\vec{t} \tag{7.2}$$

The aim is to bring M_X in the shape of an identity matrix. This means, that every left variable is no longer dependent on any other left variables and recursion is removed. When the linear equation system is solvable, a representation of the environment exists, that has no recursive dependencies.

Example 8. Let's assume the expression let $x_1 = x_2 \oplus x_3, x_2 = t_1 \oplus x_3, x_3 = x_2 \oplus t_2$ in x_1 from previous example again.

$$\begin{bmatrix} 1 & -\frac{1}{2} & -\frac{1}{2} & 0 & 0 \\ 0 & 1 & -\frac{1}{2} & -\frac{1}{2} & 0 \\ 0 & -\frac{1}{2} & 1 & 0 & -\frac{1}{2} \end{bmatrix}$$

Note that the first row of C_x and C_t yields from the assignment of $x_1 = x_2 \oplus x_3$ with the corresponding distribution $dst(x_1) = \left[\frac{1}{2}x_2, \frac{1}{2}x_3\right]$. By subtraction, x_2 and x_3 can be rearranged to the left. This causes the negative signs. Bringing the matrix C_x in form of the identity matrix with Gaussian elimination results in the following equations:

$$\begin{bmatrix} 1 & 0 & 0 & -\frac{1}{2} & -\frac{1}{2} \\ 0 & 1 & 0 & -\frac{2}{3} & -\frac{1}{3} \\ 0 & 0 & 1 & -\frac{1}{3} & -\frac{2}{3} \end{bmatrix}$$

The complete set of transformation steps applied can be found in the appendix 11. Then, C can be moved to the other side again, which swaps the sign of all t_i again. Now, the distribution of x_i is the i-th row multiplied by v. In this example, the result can be read off from the first row:

$$x_1 \sim_d \tfrac{1}{2}t_1 + t_2\tfrac{1}{2} = \left[\tfrac{1}{2}t_1, \tfrac{1}{2}t_2\right]$$

A major advantage of this method is, that the distributions of all variables are computed at once. The downsides of this procedure are definitely present in the lack of computability of the equations and in the high number of computational steps required for large environments. The latter can be kept small by using garbage-collection to shrink the size of the environment. Some systems of equations cannot be solved, since the row may vanish.

8 Distribution to Λ_{PNeedR}

In the previous section we have shown, that recursive dependencies in the environments can be removed using equation systems. The solution of the equation system is given by a set of distributions representing the definitions. It is therefore necessary to transform the distributions back into an expression. This is straight forward for the lambda-calculus Λ_{PNeedR}. The transformation into an expression of Λ_{PNeedR} is by far more complicated, like we will see in this chapter.

8.1 dst to λ_{PNeedR}

Theorem 1. *For every arbitrary, finite distribution $dst(s)$, there exists an expression $t \in \lambda_{PNeedR}$ such that $dst(t) \sim_D dst(s)$.*

Proof by induction: Let $dst(s) := [p_1 s_1, ..., p_n s_n]$ be the distribution of expression s, then the base case is a distribution with a single WEP:

$$dst(s) \quad := \quad [s] \quad \sim_d \quad s$$

For the induction step, we cut off the first element of the distribution and inserts a prob operator in between while keeping track of the probabilities.

$$\begin{aligned}
dst(s) \quad &:= \quad [p_1 s_1, ..., p_n s_n] \\
&\sim_d \quad [p_1 s_1] + [p_2 s_2, ..., p_n s_n] \\
&\sim_d \quad \tfrac{1}{p} \cdot [p_1 s_1] \oplus_{p_1} \tfrac{1}{p-1} \cdot [p_2 s_2, ..., p_n s_n] \\
&\sim_d \quad \tfrac{1}{p_1} \cdot [p_1 s_1] \oplus_{p_1} \tfrac{1}{1-p_1} \cdot [p_2 s_2, ..., p_n s_n] \\
&\sim_d \quad s_1 \oplus_{p_1} \tfrac{1}{1-p_1} \cdot [p_2 s_2, ..., p_n s_n] \\
&\sim_d \quad s_1 \oplus_{p_1} [\tfrac{p_2}{1-p_1} s_2, ..., \tfrac{p_n}{1-p_1} s_n]
\end{aligned}$$

The second sub-expression of the prob-operator is again a distribution on which the induction applies. Since the size of the distribution shrinks by one element in each iteration, the number of iteration is finite. □

This proof tells us that for every finite distribution, there must be a non-recursive program in Λ_{PNeedR}. These programs can be calculated with operational semantics in a very short time. Due to induction, this also showed, that it suffices to only prove the existence of a prob-expression for the first inserted $\oplus$-operator. In the next sections we show that under some minor restrictions, an expression in Λ_{PNeed} can be found for every finite distribution.

8.2 λ_{PNeedR} to λ_{PNeed}

The conversion from distributions into λ_{PNeed} is much more complex.

Theorem 2. *For every arbitrary, finite distribution $dst(s) = [(p_i s_i)|i \in I]$ with $p_i \in \mathbf{Q}$, there exists an expression $t \in \lambda_{PNeed}$ such that $dst(t) \sim_D dst(s)$.*

The difficulty in proving this is to represent the arbitrary probabilities using only the $\oplus$ operator, which only supports 50/50 probabilities. We already know, that 2 holds. Then, we first transform the distribution into an expression of λ_{PNeedR}. Since the only difference of these languages are the prob-operators, it suffices to only show the simulation of $\oplus_p$ in Λ_{PNeed}. It should be said in advance, that most of these programs uses recursion and therefore the finiteness of the evaluation with operational semantics cannot apply here. To make the proof understandable, let's start the topic with an intuitive analogy.

8.2.1 k-fair players in Λ_{PNeed}

Probably everyone knows the decision problem of randomly (and fairly) choosing one out of two people who is the winner or allowed to start. The coin toss can be used as an analogy for the $\oplus$-operator since it also provides two events, head or tail, that can occur with a 50/50 probability. Head and tail can be assigned to the players. Let p_i be the i-th player. In Λ_{PNeed} this can be modelled as follows:

$$p1 \oplus p2$$

This expression evaluates randomly to p_1 when $probl$ is applied or p_2 when $probr$ is applied, where p_i encodes the winning player. The game becomes a bit more complicated when 2^n players take part of the game. Then the coin must be tossed n times, so there are 2^n many different sequences of heads or tails that can be assigned to each player.

Example 9. For $n = 3, 2^n = 8$ players take part of the game. Then the coin is tossed n times. Because the coin is tossed n times, there exists 2^n sequences $\{head, tail\}^n$ (representing the prob-sequences introduced in [2]). Each player can uniquely be assigned to a sequence. After the coin toss, the player with the observed sequence wins. This can be encoded by the following expression:

$$((p1 \oplus p2) \oplus (p3 \oplus p4)) \oplus ((p5 \oplus p6) \oplus (p7 \oplus p8))$$

But what if the number of players are not powers of two? In this case, we need to get use of recursive behavior of the Λ_{PNeed} calculus. If n players (not restricted to a power of two) play, the coin must be tossed at least k times, such that at least n sequences of heads and tails occurring after k tosses. It holds for $n \geq \lceil log_2 k \rceil$. Then the 2^n sequences are enough to allocate one sequence to each player. However, events that have not been allocated to any player may remain. If such an event is observed after the coin toss, the game is restarted. Restarting the game ensures that always one player wins and that fairness is maintained in the limit of repetitions. In order to realize this in a program, we exploit the recursive behavior of the recursive let environments and define the game of three players as follows:

$$\text{Let } rec = (p1 \oplus p2) \oplus (p3 \oplus rec) \text{ in } rec$$

Figure 8.1: Expression that evaluates to p_1, p_2, p_3 with equal probability of $p = \frac{1}{3}$

Example 10.

The term is plugged into the x simulating a restart of the whole game. A problem arises in this program. The evaluation of this term is not finite because the self application can be performed over and over. However, by intuition, the property of restarting multiple times converges to zero fast. This is a perfect example of probabilistic convergence that is described in [4]. This example shows us, that we can create probabilistic convergent programs that can fairly choose items from a set with arbitrary cardinality $k \in \mathbb{N}$. The probability of each event is given by $\frac{1}{k}$ and because k is chosen arbitrarily, it follows that any reciprocal of a natural number can be simulated in a prob-program using $\oplus_p$.

We can also give players different winning probabilities by varying the number of winning events per player. If an event is included in the set $l \in \mathbb{N}$ times, the event has a probability of $\frac{l}{k}$. Since the set has a cardinality of k, l has to be in the range of $[0, k]$.

8.2.2 s_pt to Λ_{PNeed}

The observation made in section **??** can be used to prove the following theorem.

Theorem 3. *For every arbitrary, finite distribution* $dst = [(p_i s_i)|i \in I]$ *with* $p_i \in \mathbb{Q}$, *a distribution equivalent expression in* λ_{PNeed} *can be found.*

Proof: Therefore, we show, that the operator $\oplus_p$ can be simulated by the operator $\oplus_p$. Since this is the only difference between the calculi $lambda_{PNeedR}$ and λ_{PNeed}, theorem 2 can be applied after the simulation. We now present an algorithm, that computes an expression in λ_{PNeed} for every arbitrary $\oplus_p$.
Let $rec \in \Lambda_{PNeed}$. Since $p \in \mathbb{Q}$, there is a numerator denoted as p_n and a denominator denoted as p_d. Calculate the number $d = \lceil log_2(p_d) \rceil$ and construct an expression that is made of nested $\oplus$-expressions of depth d that is a balanced binary tree in graph notation. The expressions have $n = 2^d$ sub-expressions at leaf position that denoted as $[s_1, ..., s_n]$. Then assign a subset of p_n many s_i with s, $p_d - p_n$ many with t and the remaining $n - p_d$ leaves with the recursive call rec. Since $p \in [0, 1]$ is a probability, the relations $p_n \leq p_d \leq d$ must always hold, and an assignment can always be found. Finally, we check that the probability distribution of both expressions is identical. The (wsr,$\oplus$) reduction on distributions yields $t' \xrightarrow{wsr,\oplus} [ps, (1-p)t]$. For s every s_i is a subexpression of a nested prob-expression. Applying the fair prob-expression d times $\xrightarrow{wsr,\oplus d}$ s_i results in an s_i with probability $\frac{1}{2^d} = \frac{1}{n}$ and a distribution of $[\frac{1}{n}s_1, ..., \frac{1}{n}s_n]$. We sum up all s_i that are labeled equally.

$$\begin{array}{llll}
 & dist(x) & \sim_d & [\frac{p_n}{n}\,s, \frac{p_d - p_n}{n}\,t, \frac{n-p_d}{n} x] \quad\quad | - \frac{n-p_d}{n}\,x \\
\Leftrightarrow & dist(x) - \frac{p_d}{n}\,s & \sim_d & [\frac{p_n}{n}\,s, \frac{p_d - p_n}{n}\,t] \quad\quad | \cdot \frac{n}{p_d} \\
\Leftrightarrow & dist(x) & \sim_d & [\frac{p_n}{p_d}\,s, \frac{p_d - p_n}{p_d}\,t] \\
\Leftrightarrow & dist(x) & \sim_d & [p\,s, (1-p)\,t] \\
\Leftrightarrow & dist(x) & \sim_d & s \oplus_p t
\end{array}$$

This completes the proof of theorem 3. □

Example 11.

$$s \oplus_{\frac{3}{5}} t \Rightarrow \text{let}\, rec = (((s \oplus s) \oplus (s \oplus t)) \oplus ((t \oplus rec) \oplus (rec \oplus rec)))\ \text{in}\ rec$$

By the use of distributive and idempotent transformation rules from [2] this expression could be reduced. On the resulting expression, optimizations may be applicable using idempotence or rearranging the leaves. Given an example for a distribution equivalent minimal expression, optimization is not discussed further in this work.

$$s \oplus_{\frac{3}{5}} t \Rightarrow \text{let}\, rec = ((s \oplus (s \oplus x)) \oplus (t \oplus x))\ \text{in}\ rec$$

With the methods provided so far, one can prevent for let-expression to fall into infinite reduction sequences caused by recursion. The definitions of an environment can be seen as linear equations, can be modified using linear algebra and finally can be reassembled as expressions of the calculi Λ_{PNeed} and Λ_{PneedR}.

9 Interpreter

It was of natural choice to test the methods investigated during the previous chapters by implementing a Haskell interpreter. Parts of the interpreter were taken or inspired from the interpreter used in [4] but adjusted to the context of this work. The code was developed with the Glasgow Haskell Compiler (GHCi), version 8.8.4 that was installed on a Linux machine (Linux Debian). More information about the compiler can be found under https://www.haskell.org/ghc/. The interpreter can be found online at https://gitlab.com/functional-programming2/learn-gitlab/-/tree/master. To start the interpreter via GHCi, download the project folder and move into the folder where the "main.hs" , "Distribution.hs" and "Matrix.hs" are located and run the GHCi. Then, the "main.hs" file can be loaded using the following command:

```
>> :l main.hs
```

Now, the interpreter is ready to use.

9.1 Functionalities

The functionality is extended such that recursive expressions can be computed as introduced. Lambda expressions of the calculus λ_{PNeedR} can be defined, common properties can be queried, standard reduction rules can be invoked. The weighted standard reduction is implemented, as well as the evaluation strategies using Markov chains and Gauss elimination. This functionality comes along with a set of functions that work on distributions and matrices.

9.1.1 The Calculus

The type of expressions is defined like introduced in chapter 2. The implementation is similar to the interpreter of [4] and only has slightly changes for simplification. Variables are of a fixed type String. The operator defined is the $\oplus_p$ operator provided with the bias p. Thus, the $\oplus$ operator can be represented easily by fixing $p = 0.5$. The other direction is not as trivial, as shown in section 8.

```
data Expr = Var String
          | Lam String Expr
          | App Expr Expr
          | Prob Rational Expr Expr
          | Let Env Expr
          deriving Eq

type Env = [Definition]
data Definition = Def String Expr deriving Eq
```

By deriving equality for expressions, they can be compared by definition. Environments are defined as lists that contain definitions. Definitions contains a left variable that is a string and an expression. The instance for **Show** is implemented to provide a pretty string representation when the expressions are printed to the console. This includes the usage of a UTF-8 charset for printing out the special characters, like "λ" and "$\oplus$." The bias p of the prob-operator is automatically hidden, if the probability is $\frac{1}{2}$. In case of 0 or 1, the right or the left side is omitted respectively.

Example 12. let $x_1 = t_1 \oplus x_2, x_2 = t_2 \oplus x_1$ in x_1

This expression from example 3 can be typed into the interpreter as follows.

```
Let [Def "x_1" (Prob 0.5 (Var "t1") (Var "t2)), Def "x2" (Prob 0.5 (Var "t2") (Var "x1))]
    (Var "x1")
```

Pressing enter yields in the following output:

```
(let x1 = (t1 ⊕ t2), x2 = (t2 ⊕ x1) in x1)
```

9.1.2 Variables

The functions **fv** and **bv** compute free and bound variables implementing the rules of 2.3.

```
fv :: Expr -> [String]
fv (Var x)      = [x]
fv (Lam x s)    = delete x (fv s)
fv (App s t)    = nub ((fv s) ++ (fv t))
fv (Let env s)  = nub ((fv s) ++ (envFv env)) \\ (envBv env)
fv (Prob _ s t) = nub ((fv s) ++ (fv t))

bv :: Expr -> [String]
bv (Var x)      = []
bv (Lam x s)    = nub ([x] ++ (bv s))
bv (App s t)    = nub ((bv s) ++ (bv t))
bv (Let env s)  = nub ((bv s) ++ (envBv env))
bv (Prob _ s t) = nub ((bv s) ++ (bv t))

envFv :: Env -> [String]
envFv [] = []
envFv env@((Def x s) : xs) = (nub ((fv s) ++ (envFv xs))) \\ envBv env

envBv :: Env -> [String]
envBv [] = []
envBv ((Def x s) : xs) = nub ( [x] ++ (bv s) ++ (envBv xs))
```

The variables are stored in lists. An implementation using the data type **Set** was considered, but decided against. Using list simplifies the code because there is no need for conversion. However, this implementation is not as performant, especially due to the use of the function **nub** that deletes duplicates but has a complexity of $O(n^2)$.
The function **lv** can be invoked to get the left variables of an environment.

```
lv :: Env -> Set.Set String
lv []               = Set.empty
lv ((Def x s) : xs) = Set.insert x (lv xs)
```

Whenever a reduction is performed, care must be taken to the names of variables. The standard reduction rules (sr,cp-in) and (sr,cp-e) may copy abstractions into a redex, where the variable names can still occur. Thus, free variables can be captured erroneously. To prevent this, the function **rename** assigns every bound variable a unique variable name. For more information, see [4].

9.1.3 Distributions

The type of weighted expression pairs **WEP** and distributions **Dst** are defined to simplify the type annotations.

```
type WEP = (Rational, Expr)
type Dst = [WEP]
```

Since the probabilism requires the usage of distributions, some helpful functions are added. The function **dst** takes an expression and decomposes it recursively as far as possible using the set of rules 4.3.

```
dst :: Expr -> Dst
dst (Var x)      = [(1,(Var x))]
dst (Lam x s)    = [(1,(Lam x s))]
dst (App s t)    = dstAppDst (dst s) (dst t)
dst (Prob p s t) = dstAddDst (skalarMulDst p (dst s)) (skalarMulDst (1-p) (dst t))
dst (Let e s)    = [(1,(Let e s))]
```

If an expression only contains prob-expressions and variables, the environment recursion vanishes always.
Many other functions have been implemented that are not listed here for reasons of space. To group equal expressions within the distribution, **dstNub** rebuilds the distribution by successive add a WEP (p, s) while observing, whether a WEP (q, s) is contained. If yes, the probabilities are added such that $(p + q, s)$ is in the distribution. Alternatively, the WEP is added as a new element.

```
dstNub :: Dst -> Dst
dstNub [] = []
dstNub (wep@(p,s) : xs)
    | dstIsElem xs s = dstNub (dstAddWep xs wep)
    | otherwise = wep : (dstNub xs)
```

The function **`dstClean`** removes all the WEPs that are included in the distribution but have a probability of zero. This function is invoked from the function **`rewriteGauss`**, where the distributions are reassembled from the matrix and contain unnecessary entries.

```
dstClean :: [(Rational,Expr)] -> [(Rational,Expr)]
dstClean [] = []
dstClean (wep@(p,s) : xs) = if (p == 0 || p < 0) then (dstClean xs) else wep : (dstClean
    xs)
```

`rewriteMarkov` uses a similar function **`cleanRound`**. It also removes weighted terms that fall below a certain threshold. The threshold is taken as an additional parameter.

9.1.4 Reduction

The function **`srReduce`** implements the standard reduction rules 2.1. The implementation is taken from [4] and adjusted for our case. It takes an expression and tries to apply one of the standard reduction rules of [2] while traversing the expressions in the order of reduction contexts. If no reduction is applicable, a **`Nothing`** is returned. If a prob-reduction is feasible, it returns a **`Branch s' t'`**, where **`s'`** and **`t'`** are the reduced expressions respectively. For any other reduction, a **`Just s'`** with the reduced expression **`s'`** is returned.

```
stReduce :: Expr -> Maybe (Result Expr)
```

The prob-operator can not be implemented using pure functional Haskell, since probabilism is required. But it can be simulated by performing a weighted reduction and then sample from the distribution.

9.1.5 Evaluation

The function **`evaluate`** is the heart of the interpreter. On the one hand, it lifts the standard reduction to a reduction on distributions, on the other hand, the functions **`rewriteGauss`**, **`rename`** and **`garbageCollection`** are applied after every reduction step. The different strategies of evaluation are implemented. The function **`evaluateWSR`** takes an expression and tries to compute the probability distribution by only applying weighted standard reduction steps.

```
dstGauss :: Dst -> [String] -> Dst
dstGauss [] _ = []
dstGauss ((p,s) : xs) vars =
    let s2 = rewriteGauss s
        (s', vars') = rename s2 vars
    in case srReduce s' of
        Nothing -> [(p, garbageCollection s')]
        Just (Next s'') -> evaluate [(p, (garbageCollection s''))] vars'
        Just (Branch q el er)
            | p >= 0 && p <= 1 ->
                let pL = q*p
                    pR = (1-q)*p
                in  (evaluate [(pL, (garbageCollection el))] vars') ++
                    (evaluate [(pR, (garbageCollection er))] vars')
            | otherwise -> error "probabilities must be between 0 and 1"
```

```
evaluateGauss e = dstGauss [(1,e)] fresh
evaluateMarkov e n = dstMarkov [(1,e)] n
fresh = [ "x_" ++ show i | i <- [1..]]
```

The second parameter is an infinite list of fresh variable names, like shown in the last row. Note that these must be distinct to the variable in the reduced expression to avoid name clashes.
To ensure that no further naming conflicts may occur, the variable names used by the function **`rename`** must be handed to the function. This can be done by infinite lists like **`fresh`** that generates an infinite list of variable names. Make sure, that they are distinct to the set of all variables within the expression.

9.1.6 sample

The non-deterministically function **`sample`** can be used to extract an expression from an distribution taking the respective probabilities into account.

```
sample :: Dst -> IO Expr
sample d = do
    r <- randomIO :: IO Double
    return (helper d (toRational r))
        where
            helper :: Dst -> Rational -> Expr
            helper [] f = Bot
            helper ((p,s) : xs) f
                | f < p = s
                | otherwise = (helper xs ((toRational f)-p))
```

This function creates a random number between 0 and 1. The helper function divides the space into intervals, such that every entry of the distribution belongs to a range $[\sum_{k=1}^{i-1} p_k, \sum_{k=1}^{i} p_k]$. The expression of the interval, in which the random number falls, is returned. A minor issue is, that the random generator creates double values and rounds them to rationals. The probabilities can be slightly off from what expected. Since it is a random experiment anyway and the errors are negligible, the effects should not be noticeable. Applying this to a distribution simulates the evaluation of the operational semantics.

9.1.7 Markov Chains

The function **evaluateMarkov** takes the expression and a number n of iterations. This tries to compute the probability distribution using Markov chains. It uses standard reduction rules. Whenever a let-expression is in the scope of the reduction context, the environment is transformed into a transition matrix. It's power to the n is computed. A vector representing the initial state is created for each variable, which is populated with zeros and a 1 at the index of the variable. Finally, by the multiplications of the initial vectors with the matrix are computed. Afterward, the distributions of the potentials can be extracted from the resulting matrix. Using the conversion technique of section 2, the distributions can be transformed back into expressions. This hopefully unravels the recursion. Finally, the weighted standard reduction is applied.

```
rewriteMarkov :: Env -> Int -> Env
rewriteMarkov env n =
    let m = Mat.toList (markov env n)
        v = getNodeVec env
        dsts = take (length env) m
        dsts2 = map (\r -> zip r v) dsts
        clean = map Dst.clean dsts2
        expr = map dstToExpr clean
    in helper expr v
        where
            helper :: [Expr] -> [Expr] -> Env
            helper [] _ = []
            helper _ [] = []
            helper (s : ss) ((Var x) : xs) = (Def x s) : helper ss xs
```

Often, the imprecision of the technique does not vanish the recursive variables completely. They are maintained in the distributions, and the recursion is still present. To bypass this issue, one can exploit the fast convergence of the probabilities. If a probability falls below a certain threshold, it simply can round to zero, which removes the variable fully. This is exactly what the function **`cleanRound`** does. The threshold is taken as a parameter. When the evaluation does not terminate, the threshold can be increased, causing more imprecise results or the power n can be increased, Which requires more computational power.

```
cleanRound :: (Num a, Ord a) => a -> Dst a b -> Dst a b
cleanRound _ [] = []
cleanRound d ((p,s) : xs)
    | p < d = cleanRound d xs
    | otherwise = (p,s) : cleanRound d xs
```

9.1.8 Transition Matrix

The transition matrix of an environment can compute by the function **`getTransitionMatrix`**. It requests the order of nodes in the matrix, initializes an identity matrix requests the dependencies and inserts them into the matrix.

```
getTransitionMatrix :: Env -> Mat.Matrix Rational
getTransitionMatrix env =
    let v = getNodeVec env
        n = length v
        i = Mat.matrix n n Mat.identity
        edges = envGetEdges env v
    in  (setEdges edges i)
```

```
        where
            envGetEdges :: Env -> [Expr] -> [(Int, Int, Rational)]
            envGetEdges [] v = []
            envGetEdges (d : xs) v = (defGetEdges d v) ++ envGetEdges xs v
```

The vector v which stores the order of node-expression in the matrix can be requested using the function **`getNodeVec`**.

```
getNodeVec :: Env -> [Expr]
getNodeVec [] = []
getNodeVec ((Def x s) : xs) = nub ([(Var x)] ++ (getNodeVec xs) ++ (getNodes s))
```

9.1.9 Gauss Elimination

The function **`envRewriteGauss`** works similar to the function **`envRewriteMarkov`**. The distribution is computed using weighted standard reduction until a let-expression comes into the scope of a reduction context. The environment is transformed into a set of equations that can be solved with Gaussian elimination. Now, the result is still in the form of distributions belonging to the potentials. Using the conversion of 2 again writes them back to the environment. Finally, the weighted standard reduction is applied.

```
envRewriteGauss :: Env -> Env
envRewriteGauss env =
    let v = getNodeVec env
        r = length env
        c = length v
        dm = getDependentMatrix env
        iden = Mat.toIdentity dm
        inv@(Mat.Mat r1 c1 m) = Mat.mapAll (*(-1)) iden
        dsts1@(Mat.Mat r2 c2 m2) = Mat.addMatrix inv (Mat.matrix r1 c1 Mat.identity)
        dsts2 = map (\r2 -> zip r2 v) m2
        clean = map Dst.clean dsts2
        expr = map dstToExpr clean
    in helper expr v
        where
            helper :: [Expr] -> [Expr] -> Env
            helper [] _ = []
            helper _ [] = []
            helper (s : ss) ((Var x) : xs) = (Def x s) : helper ss xs
```

9.1.10 dependent Matrix

getDependentMatrix is a function that takes an environment and computes the dependent matrix. It requests the vector v of node-expressions

```
getDependentMatrix :: Env -> Mat.Matrix Rational
getDependentMatrix env =
    let v = getNodeVec env
        r = length env
        c = length v
        i = Mat.matrix r c Mat.identity
        edges = getEdges env v
    in  (setEdgesGauss edges i)
        where
            getEdges :: Env -> [Expr] -> [(Int, Int, Rational)]
            getEdges [] _ = []
            getEdges (d : xs) v = (defGetEdges d v) ++ getEdges xs v
```

First, the vector v of node-expressions is computed with the call of the function **getNodeVec**. A $n \times m$ identity matrix is defined, where n is the number of definitions i.e. the number of left-variables in the environment, and m, which is the cardinality of v. Last but not least, the edges, represented by the dependencies, are calculated and inserted into the matrix. The linear equations of the definitions are given row-wise in the final matrix.
The function **getNodeVec** extracts the list of node-expressions from the environment. It usess **getNodes**, that returns a list of leaf-nodes of a P-Context.

```
getNodes :: Expr -> [Expr]
getNodes e@(Prob p s t) = (getNodes s) ++ (getNodes t)
getNodes s = [s]

getNodeVec :: Env -> [Expr]
getNodeVec [] = []
getNodeVec ((Def x s) : xs) = nub ([(Var x)] ++ (getNodeVec xs) ++ (getNodes s))
```

9.1.11 λ_{PNeedR} to λ_{PNeed}

The function **probPToProb** takes an expression $s \oplus_p t$ of calculus λ_{PNeedR} and creates a distribution equivalent expression $s \oplus t$ in λ_{PNeed}. The function creates a p-context that represents a balanced tree of nested prob-expressions. Let d_p be the denominator of p. Then the tree must have at least d_p many leafs, i.e. the next higher power of two. Then, by the number of assigned leaves with s, t and recursive calls, the number p can be simulated.

```haskell
probPToProb :: Rational -> Expr -> Expr -> Expr
probPToProb 0  _ e2 = e2
probPToProb 1 e1 _  = e1
probPToProb p e1 e2
    | p < 0 || p > 1 = error "probPToProb: p is not in range [0,1)"
    | otherwise = fairExprs ((replicate nu e1)
                ++ (replicate (de - nu) e2)
                ++ (replicate (n - de) (Var "rec")))
        where nu = fromIntegral (numerator p)
              de = fromIntegral (denominator p)
              n = ((ceiling . logBase 2.0 . fromIntegral) de)

fairExprs :: [Expr] -> Expr
fairExprs [] = error "kFair 0 is undefined"
fairExprs exprs = exprProbTree (exprs ++ (replicate d (Var "rec")))
    where n = length exprs
          k = ((ceiling . logBase 2.0 . fromIntegral) n)
          d = (2^k) - n
```

The tree could be optimized in the count of prob expressions, since the resulting expressions often get extremely long, this function should be seen as an theoretically proof, but not as a beneficial tool.
The function "distToProbR" that converts distribution into an expression of calculus Λ_{PNeedR} looks surprisingly simple.

```haskell
distToProbR :: [(Rational,Expr)] -> Expr
distToProbR [(p,s)] = s
distToProbR ((p,s) : xs) = Prob p s (distToProbR (distMult xs (1/(1-p))))
```

9.1.12 Matrix Operations

In the functions related to Markov chains and Gauss elimination, matrix operations are used that are implemented in a separate file "Matrix.hs". The required matrix operations are implemented such as matrix multiplication, a generator for setting up identity matrices, higher order functions that apply to single rows, etc. We'd like to show the matrix multiplication and a function that calculates the power of a function, since they are used in the solving strategy using Markov chains. The multiply is implemented using the function **`zipWith`**.

```
multiply :: Num a => Matrix a -> Matrix a -> Matrix a
multiply mat1@(Mat r1 c1 m1) mat2@(Mat r2 c2 m2) =
    Mat r1 c2 [[
        sum ( zipWith (*) (getRow i mat1) (getCol j mat2))
    | j <- [0..(c2-1)] ] | i <- [0..(r1-1)] ]
```

In order to calculate the power of a matrix, the function **`power`** is implemented as a divide and conquer algorithm. The exponent n is split into half if it is even. The two resulting smaller problems are equivalent and can be computed at once. A logarithmic speedup can be observed in contrast to the naive strategy of multiplying the matrix n times. Through that, the function is still performant for large n.

```
power :: Num a => Matrix a -> Int -> Matrix a
power m 1 = m
power m n
    | n < 1           = error "power: The power of the matrix must be grater than 0"
    | n `mod` 2 == 0 = multiply (power m (n `div` 2)) (power m (n `div` 2))
    | otherwise       = multiply m (power m (n-1))
```

9.2 Testing Expressions

We tested the implementation with some of the example expressions from the previous sections. The behavior is as expected. Some of the recursive expressions will not terminate when evaluating with **`evaluate`**, but when using **`evaluateGauss`**. The recursive expression **`fair3`** is a simple recursive example that can be used to test the behavior of all three evaluation strategies.

```
>> fair3 = Let [ Def "rec"
    (Prob 0.5
        (Prob 0.5
            (Var "p1")
            (Var "p2")
        )
        (Prob 0.5
            (Var "p3")
            (Var "rec")
        )
    )
] (Var "rec")
```

Pressing enter binds the expression to the name **`fair3`**. The expression is remembered and can be invoked by calling its short name. First, the common evaluation is applied:

```
>> evaluate fair3
[(1 % 4,(let x_3 = p1 in x_3)),
 (1 % 4,(let x_3 = p2 in x_3)),
 (1 % 4,(let x_3 = p3 in x_3))
***Exception: stack overflow
```

The evaluation crashes when the recursive variable comes in scope of the reduction. The recursion does not end and the infinite sum of probabilities completely fills the storage until the exception is thrown. Running the evaluation on the interpreter of [4] will also fail with returning an error message, that the evaluation has stopped after 100 reduction steps.

```
>> evaluate fair3
[(341 % 1024,(let x_3 = p1 in x_3)),
 (341 % 1024,(let x_3 = p2 in x_3)),
 (171 % 512,(let x_3 = p3 in x_3))
***Exception: stack overflow
```

In contrast, the evaluation using Markov terminates. Even with the fifth power, the probabilities do not deviate even by a thousandth (using **`Dst.toFloat`**, the distribution is converted to float values for better readability).

```
>> evaluateMarkov fair3 5 (1/1000))
[(0.3330078125,(let x_2 = p1 in x_2)),
 (0.3330078125,(let x_3 = p2 in x_3)),
 (0.333984375,(let x_3 = p3 in x_3))]
```

Then we can apply that expression to functions like **evaluateGauss**. Pressing enter again will output the accurate distribution as shown.

```
>> evaluateGauss fair3
[(1 % 3,(let x_2 = p1 in x_2)),
 (1 % 3,(let x_3 = p2 in x_3)),
 (1 % 3,(let x_3 = p3 in x_3))]
```

All the distributions look confusing because of the renamed variables, and the environments are preserved even though garbage collection is applied. This is since the definitions might be needed in future reduction. As explained in section 8.2.1, the result is a distribution that contains three weighted expression pairs with probability $p = \frac{1}{3}$ each. Taking a closer look into the function **evaluate** that manages the steps of evaluation reveals, that the function **rewriteGauss** is applied.

```
>> rewriteGauss fair3
(let rec = (p1 ⊕1/3 (p2 ⊕ p3)) in rec)
```

The dependent matrix can be requested as well. This one consists of a single row.

```
>> getDependentMatrix (getEnv fair3)
3 % 4     (-1) % 4   (-1) % 4   (-1) % 4
```

An interesting observation can be made by sequentially apply the functions **rewriteMarkov** without rounding followed by the **rewriteGauss**. For all arbitrary chosen n being the power of the transition matrix in the Markov chain, the result is identical. It turns out that the expressions of the rewriteMarkov must be mutual distribution equivalent.

9.3 Discussion

There are still a lot of expression, the interpreter still cannot compute, because the evaluation will be trapped in an infinite loop. But how can we tell if an expression is computable using the procedures we have investigated? The methods using Markov chains and Gauss require the computation of a dependent graph. Therefore, linear equations are extracted from the definitions using the rules 4.3. The dependencies that can be noticed are those, that occur as single variables in the distribution of the Definitions. But applications, abstractions and let-expressions within the distribution can contain other left variables (dependencies) as well. Those are not recognized and the potential of eliminating recursion is reduced. Since the set of rules is certainly not exhaustive. The question arises whether other rules exist. By adding new rules, the number of computable distributions could increase significantly. To adjust the implementation, little effort is required, because only the **`dst`** function needs to be adjusted.

10 Conclusion

We have dealt with probability distributions and their contextually equivalent Λ_{PNeedR} expressions. It could be shown that for some distributions a non-recursive expression in Λ_{PNeedR} can be constructed, but the version in Λ_{PNeed} requires recursion. Evaluating both expressions with the operational semantics introduced by [2] will fail for the Λ_{PNeed} expression, but may terminate for the Λ_{PNeedR} expression. In conclusion, the evaluation strategy of [2] does not suffice the requirements needed to decide equality of expressions by observing their distributions. Hence, we tried to fix the issue by analyzing environmental variables that are defined using recursive dependencies. A linear equation system can be extracted from the environments. We have investigated two algorithms to solve the equations. The former uses Markov chains and outputs an approximated solution, the latter solves the equation system accurately by using Gauss elimination. Afterwards the solution can be transformed back into an expression or may provide the desired distribution. Therefore, we investigated algorithms for transforming distributions back to expressions of λ_{PNeed} and λ_{PNeed}. It has been proven, that the transformation is always be commutable when the probabilities of the distribution are rational numbers. Because the computation of distributions is undecidable in general, a compromise was made and simplified rules for decomposing distributions were given. These rewriting rules does not recognize all dependencies. Improvements can certainly be made here in order to exploit the full potential. For this reason, the algorithms do not cover all expressions that are solvable in theory, leaving many expressions for which our analyses fail. Finally, an interpreter was developed that implements the researched methods. It's nice to mention that the observations have lived up to expectations, and some of the recursive let-expressions can now be computed automatically.

11 Appendix

Acronyms

BNF	Backus-Naur-Form
DVC	distinct variable convention
WEP	weighted expression pair
WHNF	weak head noormal form

Transformations steps of example 8

$$\begin{array}{c} I \\ II \\ III \end{array} \begin{bmatrix} 1 & -\frac{1}{2} & -\frac{1}{2} & 0 & 0 \\ 0 & 1 & -\frac{1}{2} & -\frac{1}{2} & 0 \\ 0 & -\frac{1}{2} & 1 & 0 & \frac{1}{2} \end{bmatrix} \begin{array}{l} \\ \\ +\frac{1}{2}II \end{array}$$

$$\begin{array}{c} I \\ II \\ III \end{array} \begin{bmatrix} 1 & -\frac{1}{2} & -\frac{1}{2} & 0 & 0 \\ 0 & 1 & -\frac{1}{2} & -\frac{1}{2} & 0 \\ 0 & 0 & \frac{3}{4} & -\frac{1}{4} & -\frac{1}{2} \end{bmatrix} \begin{array}{l} \\ \\ \cdot\frac{4}{3} \end{array}$$

$$\begin{array}{c} I \\ II \\ III \end{array} \begin{bmatrix} 1 & -\frac{1}{2} & -\frac{1}{2} & 0 & 0 \\ 0 & 1 & -\frac{1}{2} & -\frac{1}{2} & 0 \\ 0 & 0 & 1 & -\frac{1}{3} & -\frac{2}{3} \end{bmatrix} \begin{array}{l} +\frac{1}{2}III \\ +\frac{1}{2}III \\ \end{array}$$

$$\begin{array}{c} I \\ II \\ III \end{array} \begin{bmatrix} 1 & -\frac{1}{2} & 0 & -\frac{1}{6} & -\frac{1}{3} \\ 0 & 1 & 0 & -\frac{2}{3} & -\frac{1}{3} \\ 0 & 0 & 1 & -\frac{1}{3} & -\frac{2}{3} \end{bmatrix} \begin{array}{l} +\frac{1}{2}II \\ \\ \end{array}$$

$$\begin{array}{c} I \\ II \\ III \end{array} \begin{bmatrix} 1 & 0 & 0 & -\frac{1}{2} & -\frac{1}{2} \\ 0 & 1 & 0 & -\frac{2}{3} & -\frac{1}{3} \\ 0 & 0 & 1 & -\frac{1}{3} & -\frac{2}{3} \end{bmatrix} \begin{array}{l} +\frac{1}{2}III \\ +\frac{1}{2}III \\ \end{array}$$

Operations that can be performed to bring left square matrix into the shape of an identity matrix.

Bibliography

1. LAGO, U. D.; ZORZI, M.: Probabilistic Operational Semantics for the Lambda Calculus. *CoRR*. 2011, vol. abs/1104.0195. Available from arXiv: 1104.0195.
2. SABEL, D. et al.: A Probabilistic Call-by-Need Lambda-Calculus – Extended Version. 2022.
3. SABEL, D.; SCHMIDT-SCHAUSS, M.: *Program Equivalence in a Typed Probabilistic Call-by-Need Functional Language* [EasyChair Preprint no. 8385]. EasyChair, 2022.
4. MAIO, L.: The Probabilistic Lambda Calculus with Call-by-Need-Evaluation. 2021, pp. 1–12.
5. RAU, C. et al.: Correctness of Program Transformations as a Termination Problem. In: 2012, pp. 462–476. ISBN 978-3-642-31364-6. Available from DOI: 10.1007/978-3-642-31365-3_36.
6. KENNAWAY, J. et al.: Infinitary lambda calculus. *Theoretical Computer Science*. 1997, vol. 175, no. 1, pp. 93–125. ISSN 0304-3975. Available from DOI: https://doi.org/10.1016/S0304-3975(96)00171-5.
7. PLOTKIN, G.: LCF considered as a programming language. *Theoretical Computer Science*. 1977, vol. 5, no. 3, pp. 223–255. ISSN 0304-3975. Available from DOI: https://doi.org/10.1016/0304-3975(77)90044-5.
8. FAGGIAN, C.; ROCCA, S. R. D.: Lambda Calculus and Probabilistic Computation. *CoRR*. 2019, vol. abs/1901.02853. Available from arXiv: 1901.02853.
9. BARENDREGT, H.; KLOP, J.: Applications of infinitary lambda calculus. *Information and Computation*. 2009, vol. 207, no. 5, pp. 559–582. ISSN 0890-5401. Available from DOI: 10.1016/j.ic.2008.09.003.