The Probabilistic Lambda Calculus with Call-by-Need-Evaluation

Luca Maio

Supervisor: Prof. Dr. David Sabel

21.12.2021

Contents

Abstract

Probabilistic modelling in programming languages has seen a significant rise in recent years due to topics such as machine learning and AI becoming strongly researched and utilized fields in computer science. The probabilistic λ-calculus is a natural extension of the well known formalism by a binary operator of probabilistic choice. This concept has been researched in recent years for call-by-value and call-by-name calculi, not however in regards to call-by-need evaluation.
This thesis defines an untyped probabilistic call-by-need λ-calculus with recursive let, discusses convergence and contextual equivalence in this probabilistic context, arriving at a definition of contextual equivalence of expressions that converge with equal probability, provides an interpreter for this calculus in Haskell and gives two concrete examples of probabilistic problems, ordered choice of k out of n with duplicates, and probabilistic tree growth, representing them in the calculus, computing their reduction with the interpreter and finally comparing the call-by-value, call-by-name and call-by-need evaluation strategies in regards to the operator of probabilistic choice for these examples, observing no significant difference, if any at all, between call-by-name and call-by-need, but a significantly worse performance of call-by-value.

1 Introduction

The λ-calculus is a well known formalism that has been studied extensively in many different forms. Even though it uses only very few rather simple concepts like abstraction and application, it is as powerful as modern programming languages. It even constitutes the basis of functional programming as we know it today. Different variations of the calculus have been researched since its conception, e.g. typed, untyped, linear, call-by-value, call-by-name or call-by-need and even among these there are differences like recursive and non-recursive let in call-by-need.
A rather recent variation on the established concept is the probabilistic λ-calculus, which extends it by adding a construct of probabilistic choice which, in case of a binary operator, chooses one expression with probability p or another with probability $1 - p$.
This is interesting, since probabilistic programming languages have become more prominent in recent years due to the heavy growth of fields utilizing them, such as machine learning and AI. For these languages it is essential to unify traditional deterministic programming with probabilistic modelling, in order to be able to make inferences about such models. Since λ-calculus is the basis of most general purpose functional programming languages, it is only natural to make a probabilistic λ-calculus the basis for a language, unifying functional programming and probabilistic modelling.
Call-by-name and call-by-value versions of this have already been studied in [1] and [2] but to the author's knowledge there has not been any research on using call-by-need-evaluation with this approach.
In Section 2 we will define such a call-by-need calculus, with recursive let, Λ_{PNeed} based on L_{Need} of [3], which represents the minimal extension to such a calculus in order to allow for probabilistic modelling. We will then in Section 3 explore the concept of contextual equivalence in this probabilistic setting, provide an interpreter written in Haskell in Section 4 and finally in Section 5 look at two specific probabilistic problems, their representation in Λ_{PNeed} and debate whether or not the call-by-need-evaluation benefits their computation.

1.1 Motivating Example

Typical λ-calculus uses variables, abstraction and application. But how would one model something like flipping a coin? What would even be the result of such computation? While one could surely implement a pseudo-random number generator in deterministic λ-calculus, since it is Turing complete after all, this would be a painstaking task. The natural solution is expanding these three constructs by an operator $\oplus_p$ with which we can now model our coin flip:

$$coin = (Heads \oplus_{\frac{1}{2}} Tails)$$

Now this expression should evaluate either to $Heads$ or to $Tails$ with a 50/50 chance. We will use a concept from probability theory for this: discrete probability distributions. Instead of actually choosing a result we calculate all possible results with the corresponding probability deterministically. We could then later sample from this distribution if needed. With this method we now know that the expression $coin$ should evaluate as follows:

$$coin \rightsquigarrow \{\frac{1}{2}Heads, \frac{1}{2}Tails\}$$

Such a reduction can in general encounter problems with confluence. This has been addressed in [1] for call-by-value and call-by-need and will not be topic of this thesis. We will in the next section formalize this reduction.

2 Call-by-Need Probabilistic λ-calculus Λ_{PNeed}

The calculus Λ_{PNeed} will extend the calculus L_{Need} of [3] by the binary probabilistic operator $\oplus_p$ and reduction rules for expressions containing this operator.

2.1 Preliminaries

Expressions e, e_i and environments Env of Λ_{PNeed} are defined as follows

$$e ::= w \mid \lambda w.e \mid (e_1\ e_2) \mid \texttt{letrec}\ Env\ \texttt{in}\ e \mid e_1 \oplus_p e_2$$
$$Env ::= w_1 = e_1, ..., w_n = e_n$$

where w, w_i are variables and $0 < p < 1$. We call $\lambda w.e$-expressions lambda-expressions or abstractions, $(e_1\ e_2)$-expressions applications, ($\texttt{letrec}\ Env\ \texttt{in}\ e$)-expressions letrec-expressions and $(e_1 \oplus_p e_2)$-expressions $\oplus$-expressions or prob-expressions.
A variable w is *bound* in e if, and only if, there exists $\lambda w.e$ or $\texttt{letrec}\ w = e', Env\ \texttt{in}\ e$, with w being the *binder* of all occurrences in e and Env. Occurrences of w outside of this scope are not bound by this, but may be bound by a different binder.
All variables that are not bound are called *free variables*. We denote the set of free variables of an expression e by $FV(e)$. We call an expression *closed*, or a program, if $FV(e) = \emptyset$. A variable may have both free and bound occurrences, for example x has both free and bound occurrences in $e ::= ((\lambda x.x)\ x)$.
We will consider two expressions e_1, e_2 to be *α-equivalent* if, and only if, all bound variables in e_1, e_2 can be renamed so that both expressions are definitionally equal. Such a renaming of bound variables is called an α-renaming.
The *distinct variable convention (DVC)* states that different binders always bind different variables, and the names of all free variables and binders are distinct from one-another. It is always possible to satisfy the DVC through renaming of bound variables. We will from here on out always implicitly do α-renamings when the DVC is violated, not actually writing them for simplicity, such that the DVC holds. We do not distinguish between α-equivalent expressions.

2.2 Church encodings

The calculus Λ_{PNeed} is untyped, thus typical arithmetic is not possible. To be able to mimic this in λ-calculus, Alonzo Church proposed an encoding for natural numbers in [4] that also allows usual arithmetic. With this encoding, the representation for a natural number n is the n-fold application of a function to a variable, encoded in λ-calculus as

$$n ::= \lambda f.\lambda x.f^n(x)$$

which yields

$\texttt{zero} ::= \lambda f.\lambda x.x$
$\texttt{one} ::= \lambda f.\lambda x.fx$
$\texttt{two} ::= \lambda f.\lambda x.f(fx)$
...

For addition we first define a successor function, which adds one function application, and then define addition using that function:

$$\texttt{succ} ::= \lambda n.\lambda f.\lambda x.f\ (n\ f\ x)$$

$$\texttt{plus} ::= \lambda m.\lambda n.m\ \texttt{succ}\ n$$

An example of the application of these functions can be seen in example 2.3.4.
We will also use this encoding in Section 5.2. It is also possible to encode tuples in a similar way, which we will explore further in Section 5.1.
Multiplication and the predecessor can also be defined, with the later being a bit more complex:

$$\texttt{mult} ::= \lambda n.\lambda n.\lambda f.m(nf)$$

$$\texttt{pred} ::= \lambda n.\lambda f.\lambda x.n\ (\lambda g.\lambda h.h\ (g\ f))\ (\lambda u.x)\ (\lambda u.u)$$

They are used in example 3.2.5.

2.3 Reduction

We define three different kinds of contexts: general contexts C, application contexts A and reduction contexts R

$$C ::= [\cdot] \mid (C\ e) \mid (e\ C) \mid \lambda x.C \mid \texttt{letrec}\ Env\ \texttt{in}\ C$$
$$\mid \texttt{letrec}\ \{w_i = C_i[w_{i+1}]\}_{i=1}^{m-1}, w_m = C_m, Env\ \texttt{in}\ C_0[w_1]$$
$$A ::= [\cdot] \mid (A\ e)$$
$$R ::= A \mid \texttt{letrec}\ Env\ \texttt{in}\ A \mid \texttt{letrec}\ \{w_i = A_i[w_{i+1}]\}_{i=1}^{m-1}, w_m = A_m, Env\ \texttt{in}\ A_0[w_1]$$

A context is an expression, as defined above, with a hole, denoted by $[\cdot]$, being placed inside according to the construction of the specific context. An expression e can fill the hole of a context C, which we denote by $C[e]$.
For the reduction of expressions without $\oplus_p$ we use the standard reduction $\xrightarrow{sr}$ with the following rules as laid out in [3]:

(sr,lbeta)	$R[((\lambda w.e_1)e_2)] \to R[\texttt{letrec}\ w = e_2\ \texttt{in}\ e_1]$
(sr,lapp)	$R[(\texttt{letrec}\ Env\ \texttt{in}\ e_1)e_2] \to R[\texttt{letrec}\ Env\ \texttt{in}\ (e_1\ e_2)]$
(sr,cp-in)	$\texttt{letrec}\ \{w_i = w_{i+1}\}_{i=1}^{m-1}, w_m = \lambda w.e, Env\ \texttt{in}\ A[w_1]$
	$\to \texttt{letrec}\ \{w_i = w_{i+1}\}_{i=1}^{m-1}, w_m = \lambda w.e, Env\ \texttt{in}\ A[\lambda w.e]$
(sr,cp-e)	$\texttt{letrec}\ \{w_i = A_i[w_{i+1}]\}_{i=1}^{m-1}, w_m = A_m[v_1], \{v_j = v_{j+1}\}_{j=1}^{n-1}, v_n = \lambda w.e, Env\ \texttt{in}\ A[w_1]$
	$\to \texttt{letrec}\ \{w_i = A_i[w_{i+1}]\}_{i=1}^{m-1}, w_m = A_m[v_1], \{v_j = v_{j+1}\}_{j=1}^{n-1}, v_n = \lambda w.e, Env\ \texttt{in}\ A[\lambda w.e]$
	where $A_m \neq [\cdot], m \geq 1, n \geq 1$
(sr,llet-in)	$\texttt{letrec}\ Env_1\ \texttt{in}\ \texttt{letrec}\ Env_2\ \texttt{in}\ e \to \texttt{letrec}\ Env_1, Env_2\ \texttt{in}\ e$
(sr,llet-e)	$\texttt{letrec}\ \{w_i = A_i[w_{i+1}]\}_{i=1}^{m-1}, w_m = (\texttt{letrec}\ Env_1\ \texttt{in}\ e), Env_2\ \texttt{in}\ A_0[w_1]$
	$\to \texttt{letrec}\ \{w_i = A_i[w_{i+1}]\}_{i=1}^{m-1}, w_m = e, Env_1, Env_2\ \texttt{in}\ A_0[w_1]$

Programs in weak head normal form (WHNF) are either abstractions $\lambda w.e$ or expressions $\texttt{letrec}\ Env\ \texttt{in}\ \lambda w.e$.

Example 2.3.1. *Let us consider the expression*
$CP ::= \texttt{letrec}\ x = (y\ y), y = (s\ v), s = z, z = \lambda u.u, v = z\ \texttt{in}\ x$
Here we would use (sr, cp-e), with $A = [\cdot]$, $w_1 = x$, $A_1 = ([\cdot]\ y)$, $w_2 = y$, $A_2 = ([\cdot]\ v)$, $v_1 = s$ and $v_2 = z$ and then copy $\lambda u.u$ into A_2. We will later call such assignments a context chain, meaning a sequence of assignments which are connected through A-contexts. Similarly we will call assignments satisfying the conditions of (sr, cp-in) variable chains.

Additionally we might use garbage collection in order to aid readability of expressions, as well as to reduce overhead:

(gc1) $\texttt{letrec}\ w_1 = e_1, ..., w_n = e_n, Env\ \texttt{in}\ e \to \texttt{letrec}\ Env\ \texttt{in}\ e$, if all w_i do not occur in Env, e
(gc2) $\texttt{letrec}\ w_1 = e_1, ..., w_n = e_n\ \texttt{in}\ e \to e$, if all w_i do not occur in e

Example 2.3.2. *Once fully reduced, CP from example 2.3.1 without garbage collection will look like this:*
$\texttt{letrec}\ x = u'', y = u', s = z, z = \lambda u.u, v = z, u' = v, u'' = y\ \texttt{in}\ \lambda u.u$
While this is a valid WHNF, this is not easy to read and one might not immediately know what bindings, if any, are still relevant to computation. Therefore we can use (gc2) in this case to remove all bindings, since $\lambda u.u$ no longer contains any of the variables in the environment and they can thus never become relevant again. This leaves us with just $\lambda u.u$ as the result, which anyone familiar with λ-calculus can easily interpret at first glance.

Multi-distributions are defined as follows, similar to the definition in Section II of [1]:

Definition 2.3.3.(multi-distribution) We call a multi-set $\mathtt{m} = [p_i e_i | i \in I]$, where the index set I ranges over the elements of $\mathtt{m}$, $p_i \in]0, 1]$ and e_i expressions, a *multi-distribution*, if $\sum_{i \in I} p_i \leq 1$.
We define the sum of multi-distributions $\mathtt{m}_1 + \mathtt{m}_2$ as the concatenation of lists.
We define the product of a scalar $q \in]0, 1]$ and a multi-distribution $\mathtt{m}$, $q \cdot \mathtt{m}$ pointwise: $q \cdot [p_1 e_1, \ldots, p_n e_n] = [(qp_1)e_1, \ldots, (qp_n)e_n]$.

Reductions of $\oplus_p$ expressions are handled with the reduction $\xrightarrow{\oplus}$ which has a multi-distribution as a result.

$$(\oplus) \quad R[e_1 \oplus_p e_2] \to [pR[e_1], (1-p)R[e_2])]$$

similarly to sets and multi-sets, multi-distributions can be understood as probability distributions where an event, which in this case will be an expression, can have multiple entries with different probabilities associated to it. Every probability distribution is trivially a multi-distribution, but one can convert a multi-distribution to a probability distribution through grouping of probabilities associated with equivalent events/expressions. The notion of (contextual) equivalence of expressions in Λ_{PNeed} will be discussed further in Section 3.

We further on use $\to$ as the union of $\xrightarrow{sr}$ and $\xrightarrow{\oplus}$ and lift it to multi-distributions as laid out in [1], denoting the lifting as $\Rightarrow$.

$$\frac{}{[e] \Rightarrow [e]}\ \text{L1} \quad \frac{e \to \mathtt{m}}{[e] \Rightarrow \mathtt{m}}\ \text{L2} \quad \frac{([e_i] \Rightarrow \mathtt{m}_i)_{i \in I}}{[p_i e_i | i \in I] \Rightarrow \sum_{i \in I} p_i \cdot \mathtt{m}_i}\ \text{L3}$$

where e, e_i are expressions, $\mathtt{m}, \mathtt{m}_i$ are multi-distributions.
This relation is reflexive, which is needed in order to be able to further reduce multi-distributions like $[\frac{1}{2}\lambda v.v, \frac{1}{2}((\lambda v.v)e)]$. Here part of the distribution is in WHNF, but in order to be able to apply L3, there needs to be a $\mathtt{m}_i$ such that $e_i \Rightarrow \mathtt{m}_i$ for every e_i in the multi-distribution. For expressions in WHNF the only possibility for this is through L1. With this lifting however, it is possible to always only reduce one part of a multi-distribution, even if the rest is not yet in WHNF.

This means for expressions leading to multi-distribution which contain expressions with non-terminating reduction, even paths that only require finitely many reduction steps are not necessarily fully reduced. To combat this [1] also introduces the notion of a *full lifting*, denoted by $\rightrightarrows$.

$$\frac{e \text{ is in WHNF}}{[e] \rightrightarrows [e]}\ \text{FL1} \quad \frac{e \to \mathtt{m}}{[e] \rightrightarrows \mathtt{m}}\ \text{FL2} \quad \frac{([e_i] \rightrightarrows \mathtt{m}_i)_{i \in I}}{[p_i e_i | i \in I] \rightrightarrows \sum_{i \in I} p_i \cdot \mathtt{m}_i}\ \text{FL3}$$

Through the constraint of FL1 to expressions in WHNF, now at every reduction step all elements of a multi-distribution must be reduced if possible. We will halt reduction if all elements of a multi-distribution are in WHNF.
It should also be noted, that this lifting does not strictly function as a one-step-reduction, since multiple reductions need to be made in case of more than one expression not being in WHNF, although for every individual expression only one reduction step at a time is applied.
The behaviour of multi-distributions and these liftings in regard to asymptotic evaluation and standardization are discussed in-depth in [1], specifically in Sections VI and VII. We will from now on only use $\to$ as an overloaded operator, denoting the union of standard reduction and $\oplus$-reduction, as well as its lifting, since the lifting's distinct behaviour is not a focus of this thesis.

Example 2.3.4. *For this example we will do a full reduction of a more complex expression. We will reduce* $M ::= ((\mathtt{plus}(\mathtt{one} \oplus_{\frac{1}{2}} \mathtt{two}))(\mathtt{zero} \oplus_{\frac{1}{2}} \mathtt{one}))$*, with the Church encodings as laid out above, which should intuitively reduce to the probability distribution* $\{\frac{1}{4}\mathtt{one}, \frac{1}{2}\mathtt{two}, \frac{1}{4}\mathtt{three}\}$.
Reduction yields the following:

$M \overset{(sr,lbeta)}{\to} \mathtt{letrec}\ m = (\mathtt{one} \oplus_{\frac{1}{2}} \mathtt{two})\ \mathtt{in}\ ((\lambda n.(m\ \mathtt{succ}\ n)))((\mathtt{zero} \oplus_{\frac{1}{2}} \mathtt{one}))$

$\overset{(sr,lbeta)}{\to} \mathtt{letrec}\ m = (\mathtt{one} \oplus_{\frac{1}{2}} \mathtt{two})\ \mathtt{in}\ (\mathtt{letrec}\ n = (\mathtt{zero} \oplus_{\frac{1}{2}} \mathtt{one})\ \mathtt{in}\ (m\ \mathtt{succ}\ n))$

$\overset{(sr,llet-in)}{\to} \mathtt{letrec}\ m = (\mathtt{one} \oplus_{\frac{1}{2}} \mathtt{two}), n = (\mathtt{zero} \oplus_{\frac{1}{2}} \mathtt{one})\ \mathtt{in}\ (m\ \mathtt{succ}\ n)$

$\overset{(\oplus)}{\to} [\frac{1}{2}\mathtt{letrec}\ m = \mathtt{one}, n = (\mathtt{zero} \oplus_{\frac{1}{2}} \mathtt{one})\ \mathtt{in}\ (m\ \mathtt{succ}\ n), \frac{1}{2}\mathtt{letrec}\ m = \mathtt{two}, n = (\mathtt{zero} \oplus_{\frac{1}{2}} \mathtt{one})\ \mathtt{in}\ (m\ \mathtt{succ}\ n)]$

$\overset{(sr,cp-in)+(gc1)}{\to} [\frac{1}{2}\mathtt{letrec}\ n = (\mathtt{zero} \oplus_{\frac{1}{2}} \mathtt{one})\ \mathtt{in}\ (\mathtt{one}\ \mathtt{succ}\ n), \frac{1}{2}\mathtt{letrec}\ n = (\mathtt{zero} \oplus_{\frac{1}{2}} \mathtt{one})\ \mathtt{in}\ (\mathtt{two}\ \mathtt{succ}\ n)]$

$\overset{(sr,lbeta)}{\to} [\frac{1}{2}\mathtt{letrec}\ n = (\mathtt{zero} \oplus_{\frac{1}{2}} \mathtt{one})\ \mathtt{in}\ (\mathtt{letrec}\ f = \mathtt{succ}\ \mathtt{in}\ ((\lambda x.fx)\ n)),$

$\frac{1}{2}\mathtt{letrec}\ n = (\mathtt{zero} \oplus_{\frac{1}{2}} \mathtt{one})\ \mathtt{in}\ (\mathtt{letrec}\ f = \mathtt{succ}\ \mathtt{in}\ ((\lambda x.f(fx))\ n))]$

$\overset{(sr,llet-in)}{\to} [\frac{1}{2}\mathtt{letrec}\ n = (\mathtt{zero} \oplus_{\frac{1}{2}} \mathtt{one}), f = \mathtt{succ}\ \mathtt{in}\ ((\lambda x.fx)\ n), \frac{1}{2}\mathtt{letrec}\ n = (\mathtt{zero} \oplus_{\frac{1}{2}} \mathtt{one}), f = \mathtt{succ}\ \mathtt{in}\ ((\lambda x.f(fx))\ n)]$

$\overset{(sr,lbeta)}{\to} [\frac{1}{2}\mathtt{letrec}\ n = (\mathtt{zero} \oplus_{\frac{1}{2}} \mathtt{one}), f = \mathtt{succ}\ \mathtt{in}\ (\mathtt{letrec}\ x = n\ \mathtt{in}\ (fx)),$

$\frac{1}{2}\mathtt{letrec}\ n = (\mathtt{zero} \oplus_{\frac{1}{2}} \mathtt{one}), f = \mathtt{succ}\ \mathtt{in}\ (\mathtt{letrec}\ x = n\ \mathtt{in}\ (f(fx)))]$

$\overset{(sr,llet-in)}{\to} [\frac{1}{2}\mathtt{letrec}\ n = (\mathtt{zero} \oplus_{\frac{1}{2}} \mathtt{one}), f = \mathtt{succ}, x = n\ \mathtt{in}\ (fx),$

$\frac{1}{2}\mathtt{letrec}\ n = (\mathtt{zero} \oplus_{\frac{1}{2}} \mathtt{one}), f = \mathtt{succ}, x = n\ \mathtt{in}\ (f(fx))]$

$\overset{(sr,cp-in)+(gc1)}{\to} [\frac{1}{2}\mathtt{letrec}\ n = (\mathtt{zero} \oplus_{\frac{1}{2}} \mathtt{one}), x = n\ \mathtt{in}\ (\mathtt{succ}\ x),$

$\frac{1}{2}\mathtt{letrec}\ n = (\mathtt{zero} \oplus_{\frac{1}{2}} \mathtt{one}), f = \mathtt{succ}, x = n\ \mathtt{in}\ (\mathtt{succ}\ (fx))]$

$\overset{(sr,lbeta))}{\to} [\frac{1}{2}\mathtt{letrec}\ n = (\mathtt{zero} \oplus_{\frac{1}{2}} \mathtt{one}), x = n\ \mathtt{in}\ (\mathtt{letrec}\ n' = x\ \mathtt{in}\ (\lambda f'.\lambda x'.f'(n'f'x'))),$

$\frac{1}{2}\mathtt{letrec}\ n = (\mathtt{zero} \oplus_{\frac{1}{2}} \mathtt{one}), f = \mathtt{succ}, x = n\ \mathtt{in}\ (\mathtt{letrec}\ n' = (fx)\ \mathtt{in}\ (\lambda f'.\lambda x'.f'(n'f'x')))]$

$\overset{(sr,llet-in))}{\to} [\frac{1}{2}\mathtt{letrec}\ n = (\mathtt{zero} \oplus_{\frac{1}{2}} \mathtt{one}), x = n, n' = x\ \mathtt{in}\ (\lambda f'.\lambda x'.f'(n'f'x')),$

$\frac{1}{2}\mathtt{letrec}\ n = (\mathtt{zero} \oplus_{\frac{1}{2}} \mathtt{one}), f = \mathtt{succ}, x = n, n' = (fx)\ \mathtt{in}\ (\lambda f'.\lambda x'.f'(n'f'x'))]$

$\overset{(\oplus)}{\to} [\frac{1}{2}[\frac{1}{2}\mathtt{letrec}\ n = \mathtt{zero}, x = n, n' = x\ \mathtt{in}\ (\lambda f'.\lambda x'.f'(n'f'x')), \frac{1}{2}\mathtt{letrec}\ n = \mathtt{one}, x = n, n' = x\ \mathtt{in}\ (\lambda f'.\lambda x'.f'(n'f'x'))],$

$\frac{1}{2}[\frac{1}{2}\mathtt{letrec}\ n = \mathtt{zero}, f = \mathtt{succ}, x = n, n' = (fx)\ \mathtt{in}\ (\lambda f'.\lambda x'.f'(n'f'x')),$

$\frac{1}{2}\mathtt{letrec}\ n = \mathtt{one}, f = \mathtt{succ}, x = n, n' = (fx)\ \mathtt{in}\ (\lambda f'.\lambda x'.f'(n'f'x'))]]$

$\to [\frac{1}{4}\texttt{letrec } n = \texttt{zero}, x = n, n' = x \texttt{ in } (\lambda f'.\lambda x'.f'(n'f'x')),$

$\frac{1}{4}\texttt{letrec } n = \texttt{one}, x = n, n' = x \texttt{ in } (\lambda f'.\lambda x'.f'(n'f'x')),$

$\frac{1}{4}\texttt{letrec } n = \texttt{zero}, f = \texttt{succ}, x = n, n' = (fx) \texttt{ in } (\lambda f'.\lambda x'.f'(n'f'x')),$

$\frac{1}{4}\texttt{letrec } n = \texttt{one}, f = \texttt{succ}, x = n, n' = (fx) \texttt{ in } (\lambda f'.\lambda x'.f'(n'f'x'))]$

The final multi-distribution only contains expressions in WHNF so no more reductions are possible. If we however look at what the expressions represent, we see that if we were to apply a call-by-value like reduction, using C-contexts in (sr), to these expressions they would reduce to **one**, **two**, **two** *and* **three** *respectively, which mean that intuitively the second and third expression in the multi-distribution should somehow be viewed as equivalent, which yields the above probability distribution.*

Since we will later also want to compare the call-by-need strategy with the call-by-name and call-by-value strategies, we will introduce analogue reduction rules to $(\oplus)$:

(name,$\oplus$)	$R[e_1 \oplus_p e_2] \to [pR[e_1], (1-p)R[e_2])]$, iff the hole in R is not inside of an environment.
(value,$\oplus$1)	$R[e_1 \oplus_p e_2] \to [pR[e_1], (1-p)R[e_2])]$, iff (value,$\oplus$2) and (value,$\oplus$3) are not applicable.
(value,$\oplus$2)	$R[e_1 \oplus_p e_2] \to R[e_1' \oplus_p e_2]$, iff $e_1 \to e_1'$.
(value,$\oplus$3)	$R[e_1 \oplus_p e_2] \to R[e_1 \oplus_p e_2']$, iff (value,$\oplus$2) is not applicable and $e_2 \to e_2'$.

If we choose the call-by-name strategy, at (sr,cp-in) and (sr,cp-e) we will also copy a $\oplus_p$-expression, not just abstractions, which we will denote by (name,cp-in) and (name,cp-e). Additionally copies will use substitution instead.
The (value,$\oplus$) rules will always first fully reduce the left and right expressions of a $\oplus_p$-expression before evaluating the $\oplus_p$.

Example 2.3.5. *We will evaluate the expression* $S ::= (\lambda y.y\ y)((\lambda u.u) \oplus_{\frac{1}{2}} \Omega)$, *where* $\Omega ::= (\lambda x.x\ x)(\lambda x.x\ x)$. *In all cases we first apply (sr,lbeta):*
$S \to \texttt{letrec } y = ((\lambda u.u) \oplus_{\frac{1}{2}} \Omega) \texttt{ in } (y\ y) =:: S'$
From here the three reduction strategies differ. Representing call-by-need, $(\oplus)$ *will immediately evaluate the* $\oplus_{\frac{1}{2}}$ *and then share it over the expression.*
$S' \overset{(\oplus)}{\to} [\frac{1}{2}\texttt{letrec } y = (\lambda u.u) \texttt{ in } (y\ y), \frac{1}{2}\texttt{letrec } y = \Omega \texttt{ in } (y\ y)]$
These two paths will then be further reduced, with the first one resulting in $\lambda u.u$, *while the second path will not terminate as* Ω *will have to be reduced within the environment and this expression famously does not terminate. The probability of termination is* $\frac{1}{2}$.
With the call-by-name strategy, we will first need to do a (name,cp-in)-reduction:
$S' \overset{(name,cp-in)+(gc2)}{\to} (((\lambda u.u) \oplus_{\frac{1}{2}} \Omega)\ ((\lambda u.u) \oplus_{\frac{1}{2}} \Omega))$
Now we can use (name, $\oplus$*) in function position.*
$\overset{(name,\oplus)}{\to} [\frac{1}{2}((\lambda u.u)\ ((\lambda u.u) \oplus_{\frac{1}{2}} \Omega)), \frac{1}{2}(\Omega\ ((\lambda u.u) \oplus_{\frac{1}{2}} \Omega))]$
The first path will split into $\lambda u.u$ *and* Ω, *with each having probability* $\frac{1}{4}$. *The second path needs to evaluate* Ω *before applying* $(name, \oplus)$ *again, so it does not terminate at all. The probability of termination is* $\frac{1}{4}$.
We can see that the results of call-by-need and call-by-name differ significantly, as the sharing of $\oplus_p$*-expressions models the choice being made only once, while substitution before evaluating models two independent choices.*
Lastly the call-by-value case needs to be considered. Here we will use (value, $\oplus$*3), since we need to do so before being able to apply (value,* $\oplus$*1), which results in* Ω *having to be computed which leads to the reduction never terminating.*

3 Contextual Equivalence for Λ_{PNeed}

In this section we will be discussing the notion of contextual equivalence in the calculus Λ_{PNeed}. Equivalence or equality are based on the intuitive concept, that if two things share the exact same properties, they should somehow be viewed as equivalent. If one goes to formalise this intuition, this notion of "sharing the same properties" is the key point to define. Hence there are a few different approaches as to what defines equivalence on expressions of λ-calculi.

3.1 Contextual Equivalence on Deterministic Calculi

Before we try to expand this notion to the probabilistic calculus Λ_{PNeed}, we first define contextual equivalence in terms of typical, deterministic calculi. We will use the definition of [5]:

Definition 3.1.1.(deterministic contextual equivalence) Let $D = (\mathcal{E}, \mathcal{C}, \rightarrow, \mathcal{W})$ be a calculus and s, t be D-expressions. *Deterministic contextual approximation* $\leq_D^{det}$ and *deterministic contextual equivalence* $\sim_D^{det}$ are defined as:

$$\begin{array}{lll} s \leq_D^{det} t & \text{iff} & \forall C \in \mathcal{C} : C[s]\downarrow_D \Rightarrow C[t]\downarrow_D \\ s \sim_D^{det} t & \text{iff} & s \leq_D^{det} t \wedge t \leq_D^{det} s \end{array}$$

where $\mathcal{E}$ is a set of expressions, $\mathcal{C}$ is a set of contexts, $\rightarrow$ a small-step reduction relation and $\mathcal{W} \subset \mathcal{E}$ is a set of values of the calculus, and $\downarrow_D$ is defined as follows:

Definition 3.1.2.(may convergence) Let $D = (\mathcal{E}, \mathcal{C}, \rightarrow, \mathcal{W})$ be a calculus. An expression $e \in \mathcal{E}$ *may converge*, we write $e \downarrow_D$ iff there exists a reduction sequence $e \rightarrow e_1 \rightarrow ... \rightarrow e_n$ such that $e_n \in \mathcal{W}$. We denote the negation by $e \Uparrow_D$.

In general it is not easy to show contextual equivalence of two concrete expressions, due to the quantification over all contexts. It is however comparatively easy to show that two expressions are not contextually equivalent, by giving a concrete example. For example $C = ([\cdot](\lambda f.\lambda x.x))$ shows that the famous Y-combinator $\mathcal{Y} = \lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x))$ is not contextually equivalent to the expression Ω in Λ_{PNeed}, which is well defined since these expressions do not contain $\oplus_p$.
$C[\mathcal{Y}] \rightarrow (\texttt{letrec}\ f = (\lambda f.\lambda x.x)\ \texttt{in}\ (\lambda x.f(x\ x))(\lambda x.f(x\ x))$
$\rightarrow (\texttt{letrec}\ f = (\lambda f.\lambda x.x), x = (\lambda x.f(x\ x))\ \texttt{in}\ (f(x\ x))$
$\rightarrow (\texttt{letrec}\ f = (\lambda f.\lambda x.x), x = (\lambda x.f(x\ x))\ \texttt{in}\ (\lambda f.\lambda x.x)(x\ x))$
$\rightarrow (\texttt{letrec}\ f = (\lambda f.\lambda x.x), x = (\lambda x.f(x\ x)), f' = (x\ x)\ \texttt{in}\ \lambda x.x) \overset{(gc2)}{\rightarrow} (\lambda x.x)$
and thus, $C[\mathcal{Y}]\downarrow_{\Lambda_{PNeed}}$. However Ω famously does not converge and in C it would have to be evaluated first. From this follows, that $\mathcal{Y}$ and Ω are not contextually equivalent.

3.2 Convergence in Λ_{PNeed}

This notion of contextual equivalence can be expanded to expressions containing $\oplus_p$ easily, just the notion of convergence, i.e. the meaning of $\downarrow_{\Lambda_{PNeed}}$ can be interpreted in different ways, due to the branching options.

The first version we will discuss is *must convergence*. This is the most restrictive concept of convergence.

Definition 3.2.1.(must convergence) Let $D = (\mathcal{E}, \mathcal{C}, \rightarrow, \mathcal{W})$ be a calculus. An expression $e \in \mathcal{E}$ *must converge*, we write $e \Downarrow_D$, iff all reduction sequences for e are finite and end in a value $e_n \in \mathcal{W}$. We denote the negation by $e \uparrow_D$

This is a natural concept that for Λ_{PNeed} encapsulates the notion of every possible path converging. It is also immediately clear that $e \Downarrow_D \Rightarrow e \downarrow_D$. The converse does not hold, since there can exist finite reduction sequences, but not all must necessarily be finite.
However there are expressions that it would feel natural for to say that they converge, even though they have infinite paths. This is especially the case for a probabilistic setting such as Λ_{PNeed}, since after every branching the possibility of the path decreases, so it may well tend to zero.

Example 3.2.2. *Let us first look at the expression* $P_1 = \texttt{one} \oplus_{\frac{1}{2}} \texttt{two}$. *We notice that* $P_1 \Downarrow_{\Lambda_{PNeed}}$ *because* $P_1 \rightarrow [\frac{1}{2}\texttt{one}, \frac{1}{2}\texttt{two}]$, *meaning both reduction sequences are finite. However for* $P_2 = \texttt{one} \oplus_{\frac{1}{2}} \Omega$ *we see that* $P_2 \uparrow_{\Lambda_{PNeed}}$, *since* $P_2 \rightarrow [\frac{1}{2}\texttt{one}, \frac{1}{2}\Omega]$ *and as we know* $\Omega \Uparrow_{\Lambda_{PNeed}}$, *so one of the two paths is infinite and hence* $P_2 \uparrow_{\Lambda_{PNeed}}$. *Another example for* $\uparrow_{\Lambda_{PNeed}}$ *is Section 5.2 where we have a possibility for the tree to continuously grow, even though that path gets less and less likely.*

As for expressions that we would naturally think should converge, but do not with this notion, we can observe $P_f = \lambda f.\lambda n.(f\ n) \oplus_{\frac{1}{2}} n$ in $P_{\mathcal{Y}} = (\mathcal{Y}\ P_f)$ `zero` and we see that the reduction path that always reduces the left option of the continuously applied P_f is infinite and never terminates, hence $P_{\mathcal{Y}} \uparrow_{\Lambda_{PNeed}}$. However for each branching, the probability of this path continuing halves, so this probability tends to zero very quickly and it would be intuitive to have $P_f \rightarrow m$, where the probability distribution associated with the multi-distribution m is $\{\texttt{zero}\}$, meaning the probability for getting the result `zero` is 1. It would be natural to define a convergence such that this holds. Not only is this intuitive, but it is also in accordance with the mathematical notion of probability, since stochastically the probability of `zero` is 1 and thus the probability of $P_{\mathcal{Y}}$ terminating is also 1.

Definition 3.2.3.(probabilistic convergence) Let $D = (\mathcal{E}, \mathcal{C}, \rightarrow, \mathcal{W})$ be a calculus. An expression $e \in \mathcal{E}$ *converges probabilistically*, we write $e \Downarrow\!\!\Downarrow_D$, iff the cumulative probability of all finite paths, which reduce to a value $e_n \in \mathcal{W}$, is equal to 1. We denote the negation by $e \Uparrow\!\!\Uparrow_D$.

From our above observations we can conclude that $P_{\mathcal{Y}} \Downarrow\!\!\Downarrow_{\Lambda_{PNeed}}$. We also do not get counter-intuitive results, for P_2 from example 3.2.2, which also does not converge probabilistically.
We can furthermore see, that $e \Downarrow_D \Rightarrow e \Downarrow\!\!\Downarrow_D$ since if there are no infinite paths, the cumulative probability of all finite paths must always be 1. The converse however does not hold, as we already saw $P_{\mathcal{Y}} \Downarrow\!\!\Downarrow_{\Lambda_{PNeed}}$ but also $P_{\mathcal{Y}} \uparrow_{\Lambda_{PNeed}}$. Thus we see that this notion of convergence expands upon the deterministic version.

Another similar concept of convergence is *should convergence*:

Definition 3.2.4.(should convergence) Let $D = (\mathcal{E}, \mathcal{C}, \to, \mathcal{W})$ be a calculus. An expression $e \in \mathcal{E}$ *should converge*, we write $e{\downarrow^{!}_{D}}$, iff for every reduction sequence $e \to e_1 \to ... \to e_n$ there exists a finite reduction sequence $e_n \to e_{n+1} \to ... \to e_m$ such that $e_m \in \mathcal{W}$. We denote the negation by $e{\uparrow^{!}_{D}}$.

Should convergence stems from non-deterministic calculi, where it encapsulates the notion of there being the possibility of termination at every point in reduction. In Λ_{PNeed} we can still use this concept, since probabilistic calculi expand non-deterministic ones by assigning a probability to branching. It is relatively easy to see, that $e \Downarrow_{\Lambda_{PNeed}} \Rightarrow e{\downarrow^{!}_{\Lambda_{PNeed}}}$, since in order for infinite paths to have probability 0, they need to always be able to reduce to a WHNF. The converse however does not hold, as the following example shows:

Example 3.2.5. *We will construct a function that can always reduce to a WHNF at every step, meaning it is should convergent, but at each iteration we add so many extra paths, that their cumulative probability is still greater than 0. We will use Church numerals as defined in Section 2.2 for this purpose, representing them with the usual symbols for natural numbers, i.e.* $0, 1, 2, \ldots$.
First we define a function
$\texttt{uniform}_i ::= \lambda x_1. \ldots \lambda x_i. x_1 \oplus_{\frac{1}{i}} (x_2 \oplus_{\frac{1}{(i-1)}} (\ldots (x_{i-1} \oplus_{\frac{1}{2}} x_i) \ldots))$
which takes i expressions and converts them into a uniformly distributed prob-expression. We also define a function $\texttt{rep} ::= \lambda i. \lambda x. i\ x\ x$ *that repeats a given input* $i + 1$ *times if i is a church numeral. Next we need a function that can be continuously called to create new paths:*
$\texttt{rec} ::= \lambda f. \lambda n. \texttt{uniform}_n\ 1\ (\texttt{rep}\ (\texttt{pred}\ (\texttt{pred}\ n))\ (f\ (\texttt{mult}\ 2\ (\texttt{pred}\ n))))$
This function takes a function f*, which will be* `rec` *itself, and a Church numeral* n *and computes the uniform distribution over* 1 *and* f *with argument* $2 \cdot (n-1)$ *in Church encoding.*
Now we can use $\mathcal{Y}$*, as defined above, to recursively call the function with*
$foo ::= \lambda n. \mathcal{Y}\ \texttt{rec}\ n$.
We can observe that $(foo\ n){\downarrow^{!}_{\Lambda_{PNeed}}}$ *for all* n*, since at every iteration we can stop with result* `one`*. If we however look at* $(foo\ 4)$ *we get the following result:*
$foo\ 4 \to^* \texttt{uniform}_4\ 1\ (foo\ 6)\ (foo\ 6)\ (foo\ 6) \to^* [\frac{1}{4}1, \frac{3}{4}(foo\ 6)]$
$\to^* [\frac{1}{4}1, \frac{3}{4}[\frac{1}{6}1, \frac{7}{6}(foo\ 10)]] \to^* [\frac{1}{4}1, \frac{3}{4}[\frac{1}{6}1, \frac{7}{6}[\frac{1}{10}1, \frac{9}{10}(foo\ 18)]]] \to^* \ldots$
where the probability for 1 *is* $\frac{1}{4} + \frac{3}{4} \cdot \frac{1}{6} + \frac{3}{4} \cdot \frac{5}{6} \cdot \frac{1}{10} + \frac{3}{4} \cdot \frac{5}{6} \cdot \frac{9}{10} \cdot \frac{1}{18} + \cdots = \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \frac{1}{32} + \cdots = \sum_{i=0}^{\infty} ((\prod_{k=0}^{i-1} \frac{x_k - 1}{x_k}) \cdot \frac{1}{x_i}) = 0.5$*, where* $x_0 = 4, x_{n+1} = 2 \cdot (x_n - 1)$*. Thus, since the cumulative probability of all finite reduction sequences that end in a WHNF, which are the ones that reduce to* 1*, is less than 1, meaning that* $(foo\ 4) \Uparrow\!\uparrow_{\Lambda_{PNeed}}$.

3.3 Contextual Equivalence on Λ_{PNeed}

With the above definitions we can now define contextual equivalence on Λ_{PNeed}. However in order for two expressions to be viewed as equivalent, they should not actually need to, even probabilistically, converge all the time, but only with the same probability.

Definition 3.3.1. Let e be a Λ_{PNeed}-expressions. We say e *converges with probability* p, we write $e \Downarrow^p$, iff the cumulative probability of all finite reduction sequences, that end in a WHNF, of e is p.

We see that probabilistic convergence is $\Downarrow^1$. This now allows us to sufficiently compare the convergence of Λ_{PNeed}-expressions that contain $\oplus_p$.

Definition 3.3.2.(contextual equivalence on Λ_{PNeed}) Let s, t be Λ_{PNeed}-expressions. *Contextual approximation* $\leq_{\Lambda_{PNeed}}$ and *contextual equivalence* $\sim_{\Lambda_{PNeed}}$ are defined as:

$$\begin{array}{lll} s \leq_{\Lambda_{PNeed}} t & \text{iff} & \forall C \in \mathcal{C} : C[s] \Downarrow^p \Rightarrow C[t] \Downarrow^q, \text{ where } p \leq q \\ s \sim_{\Lambda_{PNeed}} t & \text{iff} & s \leq_{\Lambda_{PNeed}} t \wedge t \leq_{\Lambda_{PNeed}} s \end{array}$$

This means we regard two expressions in Λ_{PNeed} to be contextually equivalent if, and only if, they probabilistically converge with the same probability in all contexts.

Example 3.3.3. *Let us now see an example for two expressions that are contextually equivalent in this new sense. Let it be noted, that this is not a formal proof, since quantifying over all contexts requires work beyond the scope of this thesis, but more so an intuition as to why this is most likely the case.*
We regard the expressions $s ::= e_l \oplus_{\frac{1}{2}} e_r$ *and* $t ::= e_r \oplus_{\frac{1}{2}} e_l$*. The reason we would expect these expressions to be equivalent is, that they both reduce to* $[\frac{1}{2}e_l, \frac{1}{2}e_r]$.
We assume $\forall C \in \mathcal{C} : C[e_1 \oplus_p e_2] \Downarrow^q \Leftrightarrow (C[e_1] \Downarrow^l \wedge C[e_2] \Downarrow^r)$ *holds, with* $q = pl + (1-p)r, 0 < p < 1, 0 \leq l, r \leq 1$*, where* $\Downarrow^0$ *means* $\Uparrow_{\Lambda_{PNeed}}$*, without proving this.*
We then consider four cases for an arbitrary, fixed context C*:*
Case $C[e_l] \Downarrow^p$, $C[e_r] \Downarrow^q$, $0 < p, q \leq 1$*:*
With the property above we get $C[s] \Downarrow^{\frac{1}{2}p+\frac{1}{2}q}$ *and* $C[t] \Downarrow^{\frac{1}{2}q+\frac{1}{2}p}$*, meaning they both converge with equal probability, since* $\frac{1}{2}p + \frac{1}{2}q = \frac{1}{2}(p+q) = \frac{1}{2}(q+p) = \frac{1}{2}q + \frac{1}{2}p$.
Case $C[e_l] \Downarrow^p$, $C[e_r] \Downarrow^0 \Leftrightarrow C[e_r] \Uparrow_{\Lambda_{PNeed}}$, $0 < p \leq 1$*:*
With the property above we get $C[s] \Downarrow^{\frac{1}{2}p+\frac{1}{2}\cdot 0}$ *and* $C[t] \Downarrow^{\frac{1}{2}\cdot 0+\frac{1}{2}p}$*, meaning they both converge with equal probability* $\frac{1}{2}p$.
Case $C[e_l] \Downarrow^0 \Leftrightarrow C[e_l] \Uparrow_{\Lambda_{PNeed}}$, $C[e_r] \Downarrow^p$, $0 < p \leq 1$*:*
Analogous to the previous case.
Case $C[e_l] \Downarrow^0 \Leftrightarrow C[e_l] \Uparrow_{\Lambda_{PNeed}}$, $C[e_r] \Downarrow^0 \Leftrightarrow C[e_r] \Uparrow_{\Lambda_{PNeed}}$*:*
With the property above we get $C[s] \Downarrow^{\frac{1}{2}\cdot 0+\frac{1}{2}\cdot 0}$ *and* $C[t] \Downarrow^{\frac{1}{2}\cdot 0+\frac{1}{2}\cdot 0}$*, meaning both converge with equal probability* 0.
Thus we get $s \sim_{\Lambda_{PNeed}}$ *as desired.*
We can also now verify that P_1 *and* P_2 *from example 3.2.2 are not contextually equivalent. We regard the empty context* $C = [\cdot]$*, then* $C[P_1] \Downarrow^1$*, but* $C[P_2] \Downarrow^{\frac{1}{2}}$ *and thus* $P_1 \not\leq_{\Lambda_{PNeed}} P_2$*, since* $1 \not\leq \frac{1}{2}$.

4 An Interpreter for Λ_{PNeed}

Due to this calculus following call-by-need-evaluation, the natural choice was to implement the interpreter in Haskell.
The full code is available at `https://gitlab2.cip.ifi.lmu.de/maio/BA-Prog`.

4.1 Expressions and Renaming

The basis of the calculus is implemented in a new data type `Expr a` which represents expressions of Λ_{PNeed} as one would expect.

```
data Expr a = Var a
            | Lambda a (Expr a)
            | Application (Expr a) (Expr a)
            | Letrec [Assignment a] (Expr a)
            | Prob (Expr a) (Expr a) Rational
            | RM deriving Eq
```

The `RM` constructor is needed only for the full evaluation of Church numerals, which is discussed in further detail in Section 5.2.
`Assignment a` is also as expected:

```
data Assignment a = Assig a (Expr a) deriving (Eq,Show)
```

The instance for `Show (Expr a)` is defined in a way that makes these expressions more natural to read. We also use a type-wrapper `newtype StringExpr = SE String` in order to remove quotation marks. So

```
Application
    (Lambda (SE "x") (Application (Var (SE "x"))
                                  (Var (SE "x"))))
    (Prob (Lambda (SE "x") (Application (Var (SE "x"))
                                        (Var (SE "x"))))
        (Letrec [Assig (SE "x") (Var (SE "y"))] (Application (Var (SE "x"))
                                                             (Var (SE "x"))))
                0.5)
```

becomes

```
((\x.(x x)) (prob[1 % 2] (\x.(x x)) (let x=y in (x x))))
```

which is the representation that will be used where possible in this thesis, in order to aid readability. In case strings are used we will still write them as usual, but in the implementation they will actually be a `StringExpr`.

Before every reduction step the expression is run through a rename function in order to guarantee the distinct variable convention, which takes an `Expr a`, a list of fresh variable names and returns a pair of the renamed expression and all unused fresh variable names.

```
rename :: (Eq a) => Expr a -> [a] -> (Expr a, [a])
```

The details of the renaming process are as one would expect and will be left out for the sake of brevity. The base function was taken from Section 3.1.2 of [6] and cases for `Letrec` and `Prob` were added accordingly.

4.2 Standard Reduction

4.2.1 One-Step-Reduction

The one-step-reduction is handled in the function

```
tryNeedStd :: Eq a => Expr a
                          -> Strategy
                              -> Bool
                                  -> Maybe (Result (Expr a))
```

which subsequently tries to apply one of the (sr)-rules to the given expression. As additional input it receives a `Strategy` which is either `Need`, `Name` or `Value` and is only relevant to the reduction strategy used on the `Prob` expressions.

The `Bool` argument denotes whether or not a Church numeral needs to be evaluated, which will again become relevant in Section 5.2 and will basically be handled similarly to call-by-value. This kind of reduction will however only ever be called after standard reduction has completely finished, meaning the expression is already in WHNF in the sense of call-by-need.

If a reduction is possible, a `Result (Expr a)` is returned which is defined as follows:

```
data Result a = Branch Rational a a | Next a
```

In case of a branching due to `Prob`, a `Branch` statement with the branching probability and the respective expressions are returned. In case one of the (sr)-rules apply we return a `Next` statement with the reduced expression. If no reduction is possible `Nothing` is returned and the overall reduction halts.

We first check (sr, lbeta) and (sr, lapp), which translate directly from the reduction rules:

```
tryNeedStd ((\v.e1) (e2)) _ _ =
    Just $ Next $ (let v = e2 in e1)
tryNeedStd ((let env in e1) e2) _ _ =
    Just $ Next $ (let env in (e1 e2))
```

In case of an `Application`, we follow the construction of A-contexts, meaning we try to reduce the expression in function position, for Church numerals we also check the right side, following the construction of general C-contexts:

```
tryNeedStd (e1 e2) s ch =
    case tryNeedStd e1 s ch of
        Nothing
            | not ch -> Nothing
            | otherwise ->
                case tryNeedStd e2 s ch of
                    Just (Next e2') -> Just $ Next (e1 e2')
                    Just (Branch p el er) -> Just $ Branch p (e1 el) (e1 er)
        Just (Next e1') -> Just $ Next (e1' e2)
        Just (Branch p el er) -> Just $ Branch p (el e2) (er e2)
```

For a Letrec expression, either a (cp)-reduction or an (llet)-reduction may be applicable. For the (sr, cp-in)-reduction we first check if such a reduction is possible with

```
hasVarChain :: Eq a => [Assignment a]
                            -> Expr a
                                -> Strategy
                                    -> Bool
                                        -> Bool
                                            -> Bool
hasVarChain env e s cont ch
    | not ch && hasAnyVar e =
        case getVarChain (nextVar e) env env s cont of
            Nothing -> False
            Just ex -> True
    | ch && hasAnyVarChurch e =
        case getVarChain (nextVarChurch e) env env s cont of
            Nothing -> hasVarChain env (removeVar e) s cont ch
            Just ex -> True
    | otherwise = False
```

which checks for the variable in the hole of the innermost A-context, if there is a sequence of variable-to-variable assignments such that this sequence ends in an abstraction. If the Name strategy is used, the chain can also end in a Prob statement. For Church numeral reduction this is done for every C-context, here is also where the RM constructor is used. The first Bool argument denotes if this was called in the context of hasEnvChain below.
getVarChain computes the actual search for the variable chain recursively.

```
tryNeedStd (let env in e) s ch
    | hasVarChain env e s ch ch =
        case tryNeedStd e s ch of
            Nothing -> Just $ cpin env e e
            Just (Next e') -> Just $ Next (let env in e')
            Just (Branch p el er)
                -> Just $ Branch p (let env in el) (let env in er)
```

In case the reduction is possible, i.e. (sr, cp-in) is applicable, we first try to reduce the expression inside the Letrec and if this fails we copy the found abstraction into the expression with cpin.

```
cpin :: Eq a => [Assignment a]
                    -> Expr a
                        -> Expr a
                            -> Result (Expr a)
```

This again computes the variable chain from the environment and then copies it to the corresponding place within the hole of the A-context. In case of a Church numeral reduction, it looks for the first C-context it finds a variable chain for, and then substitutes it across all C-contexts.

The next case to check is (sr, cp-e). Similarly to (sr, cp-in) here `hasEnvChain` is used to check for a chain of A-contexts, followed by a chain of variables within the given `Letrec` environment.

```
hasEnvChain :: Eq a => [Assignment a] -> Expr a -> Strategy -> Bool -> Bool
hasEnvChain env e s ch
    | not ch && hasAnyVar e =
        case getContextChain (nextVar e) env env s of
            Nothing -> False
            Just ex -> True
    | ch && hasAnyVarChurch e =
        case getContextChain (nextVarChurch e) env env s of
            Nothing -> hasEnvChain env (removeVar e) s ch
            Just ex -> True
    | otherwise = False
```

This function is nearly identical to `hasVarChain`, just calling `getContextChain` instead of `getVarChain`, which looks for the start of the possible context chain and then goes through the assignments which assign general expressions to the current variable, instead of just abstractions, iterating recursively until a variable chain after the expressions is found with `getVarChain`.

```
tryNeedStd (let env in e) s ch
    | hasEnvChain env e s ch =
        case tryNeedStd e s ch of
            Nothing -> Just $ cpe env e e
            Just (Next e') -> Just $ Next (let env in e')
            Just (Branch p el er)
                -> Just $ Branch p (let env in el) (let env in er)
```

Again we check if the inner expression can be reduced and otherwise we use `cpe` to copy the found abstraction to the corresponding place within the environment.

```
cpe :: Eq a => [Assignment a]
                    -> Expr a
                        -> Expr a
                            -> Result (Expr a)
```

The function `cpe` works similarly to `cpin`, but instead of copying the abstraction into the inner expression, it copies it into the environment accordingly.

The reduction (sr, llet-in) is handled very intuitively like (sr, lbeta) and (sr, lapp).

```
tryNeedStd (let env in (let env2 in e)) _ _ =
    Just $ Next (let env;env2 in e)
```

The last rule for the `Letrec` cases, (sr, llet-e), is handled similarly to the cp cases, with `hasEnvChainLetrec` searching for a variable chain that ends not in an abstraction, but in a `Letrec` expression, in order to combine the environments.

```
hasEnvChainLetrec :: Eq a => [Assignment a] -> Expr a -> Bool -> Bool
hasEnvChainLetrec env e ch
    | not ch && hasAnyVar e =
        case getContextChainLetrec (nextVar e) env env of
            Nothing -> False
            Just a -> True
    | ch && hasAnyVarChurch e =
        case getContextChainLetrec (nextVarChurch e) env env of
            Nothing
                -> hasEnvChainLetrec env (removeVar e) ch
            Just a
                -> True
    | otherwise = False
```

This functions in the same way as `hasEnvChain`, just disregarding the `Strategy`.

```
tryNeedStd (let env2 in e) s ch
    | hasEnvChainLetrec env2 e ch =
        case tryNeedStd e s ch of
            Nothing
                -> Just $ llete env2 e e
            Just (Next e')
                -> Just $ Next $ (let env2 in e')
            Just (Branch p el er)
                -> Just $ Branch p (let env2 in el) (let env2 in er)
```

The function `llete`, after computing the `Letrec` expression that is to be lifted, replaces the assignment containing the `Letrec` with the corresponding inner expression, and adds its environment to the overarching environment.

```
llete :: Eq a => [Assignment a]
                    -> Expr a
                        -> Expr a
                            -> Result (Expr a)
```

If none of the (cp) or (llet) cases are applicable, we next try to reduce an expression within the environment in case of a `Letrec` expression.

```
tryNeedStd (let env in e) s ch =
    case tryNeedStd e s ch of
        Nothing ->
            case tryNeedStdEnv env env s ch of
                Nothing
                    -> Nothing
                Just (Next env')
                    -> Just $ Next (let env' in e)
                Just (Branch p envl envr)
                    -> Just $ Branch p (let envl in e) (let envr in e)
        Just (Next e')
            -> Just $ Next (let env in e')
        Just (Branch p el er)
            -> Just $ Branch p (let env in el) (let env in er)
```

where tryNeedStdEnv iterates through the environment recursively and tries to reduce each expression, returning the changed environment if possible or Nothing otherwise.

```
tryNeedStdEnv :: Eq a => [Assignment a]
                              -> [Assignment a]
                                  -> Strategy
                                      -> Bool
                                          -> Maybe (Result [Assignment a])
```

Finally the ($\oplus$) case remains to be checked, which for the Name and Need strategies is computed intuitively, by returning the branching probability and both possible result expressions. However for the Value strategy we first try to reduce the expressions inside, following the (value,$\oplus$) rules, the Prob expression, starting with the left expression.

```
tryNeedStd (prob[p] a b) Value ch =
    case tryNeedStd a Value ch of
        Nothing ->
            case tryNeedStd b Value ch of
                Nothing
                    -> Just $ Branch p a b
                Just (Next b')
                    -> Just $ Next (prob[p] a b')
                Just (Branch pb bl br)
                    -> Just $ Branch pb (prob[p] a bl) (prob[p] a br)
        Just (Next a')
            -> Just $ Next $ (prob[p] a' b)
        Just (Branch pa al ar)
            -> Just $ Branch pa (prob[p] al b) (prob[p] ar b)

tryNeedStd (prob[p] a b) _ _ = Just $ Branch p a b
```

If, and only if, a Church numeral is being reduced we would also like to reduce an inner expression for a Lambda expression, again following the construction of C-contexts.
For all other possible cases we return Nothing.

```
tryNeedStd (\v.e) s True =
    case tryNeedStd e s True of
        Nothing
            -> Nothing
        Just (Next e')
            -> Just $ Next $ (\v.e')
        Just (Branch p el er)
            -> Just $ Branch p (\v.el) (\v.er)

tryNeedStd _ _ _ = Nothing
```

4.2.2 Garbage Collection

In order to actually be able to get `Lambda` expressions as WHNFs we need garbage collection as defined in Section 2.3.
This is implemented in the function `garbageCollection`, which uses `checkEnv` to iterate through the environment, and check if the assigned variables are still present anywhere within the environment or the expression, and if not, deletes that assignment. If there are no assignments left, then the whole `Letrec` is dropped and replaced with just the inner expression. Since only completely unused assignments are dropped, this can not affect the resulting reduction other than increasing efficiency on iterations through the environments slightly. `garbageCollectionEnv` calls `garbageCollection` on all expressions within the environment.

```
garbageCollection :: Eq a => Expr a -> Expr a
garbageCollection (let env in e) =
    case checkEnv env env e of
        [] -> garbageCollection e
        env'
            | env == env'
                -> (let (garbageCollectionEnv env') in (garbageCollection e))
            | otherwise
                -> garbageCollection (let env' in e)
garbageCollection (Var v) = Var v
garbageCollection (\v. e) = (\v.(garbageCollection e))
garbageCollection (e1 e2) =
    ((garbageCollection e1) (garbageCollection e2))
garbageCollection (prob[p] e1 e2) =
    (prob[p] (garbageCollection e1) (garbageCollection e2))
```

4.2.3 Multi-Step-Reduction

The multi-step-reduction is handled in the function `tryNeed`, which continuously tries one-step-reductions until `tryNeedStd` returns `Nothing`. On every step, `rename` gets called in order to guarantee the distinct variable convention.

```
tryNeed :: Eq a => Expr a
                    -> [a]
                        -> Bool
                            -> Bool
                                -> Strategy
                                    -> Rational
                                        -> Integer
                                            -> ProbTree a
```

As inputs, `tryNeed` takes the expression that is supposed to be reduced, a list of variable names for `rename`, a `Bool` argument to indicate whether or not garbage collection should be used, another `Bool` argument indicating a Church numeral is being reduced, a `Strategy` for `tryNeedStd`, the multiplied probability of the current branch in the `ProbTree` and a counter to automatically halt after a certain number of steps. The function returns a `ProbTree a` which is defined as follows:

```
data ProbTree a = PLeaf (Expr a)
                | PNode (Expr a)    (Either String (ProbTree a, Rational))
                                    (Either String (ProbTree a, Rational))
                                    deriving Eq
```

This structure encodes the full reduction process of any `Expr`. If the expression does not contain `Prob`, then the result will always be a tree consisting of only one `PLeaf`, which contains the fully reduced expression. If it does contain `Prob`, then the `PNode`'s store the reduced expression up to the point of the next one-step-reduction being ($\oplus$), and then splits into a `ProbTree` containing the left side of the `Prob` expression with the probability of the current branch multiplied by the branching probability, and a `ProbTree` containing the same for the right side of the `Prob` expression being reduced, with the probability of the current branch multiplied by 1 minus the branching probability. In case of the reduction process forcefully terminating after a certain number of steps the `PNode` has two identical branches containing an error message. For this purpose the `Either` wrapper is used.
For example the `ProbTree` generated by

```
 tryNeed (prob[(1 % 5)] one
                        (prob[(1 % 4)] two
                                       (prob[(1 % 3)] three
                                                      (prob[(1 % 2)] four
                                                                     five))))
         fresh True False Need 1 0
```

looks like this:

```
PNode (prob[(1 % 5)] one
                     (prob[(1 % 4)] two
                                    (prob[(1 % 3)] three
                                                   (prob[(1 % 2)] four
                                                                  five))))
      (PLeaf one, (1 % 5))
      (PNode (prob[(1 % 4)] two
                            (prob[(1 % 3)] three
                                           (prob[(1 % 2)] four
                                                          five)))
             (PLeaf two, (1 % 5))
             (PNode (prob[(1 % 3)] three
                                   (prob[(1 % 2)] four
                                                  five))
                    (PLeaf three, (1 % 5))
                    (PNode (prob[(1 % 2)] four
                                          five)
                           (PLeaf four, (1 % 5))
                           (PLeaf five, (1 % 5))
                    ,(2 % 5))
             ,(3 % 5))
      , (4 % 5))
```

with the `Right` constructors left out for readability. The probability of all `PLeaf`'s sums up to 1.

While this representation can be useful to gain an oversight of the whole reduction process, most of the time a probability distribution is more useful as a result. For this distribution we use the data type `ProbDistribution a` which is defined as follows:

```
type EExpr a = Either String (Expr a)
type ProbDistribution a = [(EExpr a, Rational)]
```

The function `treeToDist :: ProbTree a -> ProbDistribution a` computes this conversion as one would expect, by collecting all `PLeaf`'s into a list with their associated probabilities.
Additionally one can use

```
groupDist :: ProbDistribution String -> ProbDistribution String
```

to group the probabilities of expressions that are structurally equal modulo α-renaming.

The functions

```
reduceNeed :: Expr String -> Bool -> Strategy -> ProbTree String
reduceNeed e gc s = tryNeed e fresh gc False s 1 0
```

and

```
reduceNeedDist :: Expr String -> Bool -> Strategy -> ProbDistribution String
reduceNeedDist e gc s = groupDist $ treeToDist $ reduceNeed e gc s
```

are used to start the complete reduction of a specified `Expr String`, for which the user may also choose if they want garbage collection, and what `Strategy` they would like to be employed. `fresh` is an infinite list of variable names.

4.3 Monte-Carlo-Simulation

With the above functions one can deterministically compute the complete probability distribution (or `ProbTree`) for any given expression in Λ_{PNeed}. Depending on the expression and the maximum number of steps before stopping the calculation, this can take quite some time. On top of that it is only natural to want to have an option to only evaluate one particular path with the probabilities given in the `Prob` constructs. One could then with such a function conduct so called Monte-Carlo-simulations, where instead of computing a complete probability distribution, one simply repeats an experiment a certain number of times and calculates the probabilities of the results within these repetitions. This then gives an approximation of the probability distribution without having to actually calculate it.

Since Haskell is a functional programming language, random number generation requires monadic structures. For this implementation `System.Random` is used, specifically the function `randomRIO`, which uses the global random number generator, thus eliminating the need to input a seed every time the user starts a simulation.

The simulations are done using

```
tryNeedMCSim :: Eq a => Expr a
                             -> [a]
                                 -> Bool
                                     -> Strategy
                                         -> Integer
                                             -> IO (EExpr a)
```

which essentially functions like `tryNeed` but does not need to carry the current path's probability, and on branching generates a number between 0 and 1 with `randomRIO`. If this number is lower than the branching probability the left path is taken, the right one otherwise. Depending on the expression simulated, the counter to prevent endless computation might not be necessary if the program can still terminate after every branching, but it may still be beneficial for computation time. Also since `randomRIO` returns an `IO Double`, the results will need to be stored in the `IO`-monad as well.
The function

```
monteCarlo :: Expr String
                  -> Bool
                      -> Strategy
                          -> Integer
                              -> IO (ProbDistribution String)
monteCarlo e gc s numRep = countMC $ mCSim e gc s numRep
```

is what should be called to do a full Monte-Carlo-simulation, with `numRep` deciding the number of repetitions.
`countMC :: Eq a => IO [EExpr a] -> IO (ProbDistribution a)` simply adds the probability, being the number of times the element occurs over the length of the resulting list, to each result, while `mCSim` actually calls `tryNeedMCSim` the specified number of times.

5 Probabilistic Problems in Λ_{PNeed}

In this section we will look at two examples of probabilistic problems and how one can encode them in Λ_{PNeed}, use the function `reduceNeedDist` to fully reduce them into a probability distribution and also use the function `monteCarlo` in order to perform Monte-Carlo-simulation on these expressions. For both examples we will use the Church encodings as laid out in Section 2.2 above.

5.1 Choosing k of n with duplicates

First we will look at a classic combinatoric problem, choosing k of n with duplicates. To be precise, we will be modelling the probability distribution of all choices, accounting for order. We will use the Church numerals as our choice pool. There is also a Church encoding for pairs, which we will generalize to n-tuples, which consists of a function `pair` and the function `fst` and `snd` which each yield the first or second element of the pair respectively.

$$\texttt{pair} = \lambda a.\lambda b.\lambda f.f\ a\ b$$
$$\texttt{fst} = \lambda t.t\ (\lambda a.\lambda b.a)$$
$$\texttt{snd} = \lambda t.t\ (\lambda a.\lambda b.b)$$

The function `pair` is used to create the pair in the first place, and later `fst` and `snd` can be used to access the contents. This concept can easily be expanded to n-tuples, with functions `kthOfn` for all $k \leq n$ as follows:

$$\texttt{nTuple} = \lambda x_1.\ldots\lambda x_n.\lambda f.f\ x_1\ \ldots x_n$$
$$\texttt{kthOfn} = \lambda t.t\ (\lambda x_1.\ldots\lambda x_n.x_k)$$

In Haskell we can easily construct these with recursive functions. For `nTuple` we first construct all `Lambda` bindings, collecting all used binders in the process, and then construct the inner `Application` where we simply iterate through all saved binders and add the function application in the end. In `kthOfn`, in the k-th iteration of nesting `Lambda` expressions, we save the current binder and then use it as the inner expression in the end.

```
nTuple :: Integer -> Expr StringExpr
nTuple n = nTupRec n [] binders
    where
        nTupRec 0 collect _ =
            (\f.(makeApp collect))
        nTupRec n collect (name:bind) =
            (\name.(nTupRec (n-1) (name:collect) bind))
        makeApp [name] =
            (f name)
        makeApp (name:xs) =
            ((makeApp xs) name)
```

```
kthOfn :: Integer -> Integer -> Expr StringExpr
kthOfn k n =
    (\t.(t (kthRec k n binders [] False)))
    where
        kthRec _ 0 _ _ False =
            error "n must be between 1 and k"
        kthRec _ 0 _ [nth] _ =
            Var nth
        kthRec 1 n (name:bind) nth False =
            (\name.(kthRec 0 (n-1) bind (name:nth) True))
        kthRec _ n (name:bind) nth True =
            (\name.(kthRec 0 (n-1) bind nth True))
        kthRec k n (name:bind) nth False =
            \name.(kthRec (k-1) (n-1) bind nth False))
```

where `binders` are simply an infinite list of fresh names. Note that for each n we need to construct all choice functions and can not reuse e.g. `fst` and `snd`, since all n parts of the tuple must be processed in order for the element to be retrieved.

Now we would like a function that converts our Church encoded n-tuples into Haskell tuples. This is not possible dynamically, due to type constraints of Haskell, but we can simply use lists to mimic these tuples. This allows us to write a function `fromnTuple` which takes a `ProbDistribution`, which will be the initialized tuple that has previously been filled with values, and calls `kthOfn` for all k on it, saving each result into a list.

```
fromnTuple :: Integer
                  -> ProbDistribution StringExpr
                      -> [(Either String [Expr StringExpr], Rational)]
fromnTuple _ [] = []
fromnTuple n ((Right e, p):xs) =
    (Right (fromnTupleRec n n e), p) : fromnTuple n xs
    where
        fromnTupleRec _ 0 _ = []
        fromnTupleRec n k e =
            case reduceNeedDist ((kthOfn k n) e) True Need of
                [(Right e', p')] -> e' : fromnTupleRec n (k-1) e
                _ -> error $ "The expression" ++ show e
                             ++ "might not be a " ++ show n ++ "-tuple"
fromnTuple n ((Left s,p):xs) =
    (Left s,p) : fromnTuple n xs
```

With this we can now implement a function that actually computes the choice of k of n. We do this in the function `choosekofn`:

```
choosekofn :: Integer -> Integer -> [(Either String [Integer], Rational)]
choosekofn k n =
    let pool = makeProbExpr $ uniform $ take (fromIntegral n) allChurchNums
    in fromnTupleChurch k
           $ reduceNeedDist (choosekofnRec k k pool) True Need
```

```
    where
        choosekofnRec k 1 pool = ((nTuple k) pool)
        choosekofnRec k ctr pool = ((choosekofnRec k (ctr-1) pool) pool)
```

The list `allChurchNums` is an infinite list containing all Church numerals in order, starting from `zero`. We also use `makeProbExpr`, which takes a probability distribution, meaning a list of `Expr` paired with a probability, and constructs a matching nested `Prob` expression, as well as `uniform`, which takes a list of `Expr` and pairs each with 1 divided by the length of the list. The function `fromnTupleChurch` calls `churchToInt` on all expressions on top of the functionality of `fromnTuple`. One can also use a different function instead of `uniform` if one would like to distribute their choice pool differently, or specify a completely custom distribution altogether.

We can now compute the probability of all possible choices for specific k and n. For example `choosekofn 3 4` yields:

$[([3,0,0],\frac{1}{64}),([2,0,0],\frac{1}{64}),([1,0,0],\frac{1}{64}),([0,0,0],\frac{1}{64}),([3,1,0],\frac{1}{64}),([2,1,0],\frac{1}{64}),$
$([1,1,0],\frac{1}{64}),([0,1,0],\frac{1}{64}),([3,2,0],\frac{1}{64}),([2,2,0],\frac{1}{64}),([1,2,0],\frac{1}{64}),([0,2,0],\frac{1}{64}),$
$([3,3,0],\frac{1}{64}),([2,3,0],\frac{1}{64}),([1,3,0],\frac{1}{64}),([0,3,0],\frac{1}{64}),([3,0,1],\frac{1}{64}),([2,0,1],\frac{1}{64}),$
$([1,0,1],\frac{1}{64}),([0,0,1],\frac{1}{64}),([3,1,1],\frac{1}{64}),([2,1,1],\frac{1}{64}),([1,1,1],\frac{1}{64}),([0,1,1],\frac{1}{64}),$
$([3,2,1],\frac{1}{64}),([2,2,1],\frac{1}{64}),([1,2,1],\frac{1}{64}),([0,2,1],\frac{1}{64}),([3,3,1],\frac{1}{64}),([2,3,1],\frac{1}{64}),$
$([1,3,1],\frac{1}{64}),([0,3,1],\frac{1}{64}),([3,0,2],\frac{1}{64}),([2,0,2],\frac{1}{64}),([1,0,2],\frac{1}{64}),([0,0,2],\frac{1}{64}),$
$([3,1,2],\frac{1}{64}),([2,1,2],\frac{1}{64}),([1,1,2],\frac{1}{64}),([0,1,2],\frac{1}{64}),([3,2,2],\frac{1}{64}),([2,2,2],\frac{1}{64}),$
$([1,2,2],\frac{1}{64}),([0,2,2],\frac{1}{64}),([3,3,2],\frac{1}{64}),([2,3,2],\frac{1}{64}),([1,3,2],\frac{1}{64}),([0,3,2],\frac{1}{64}),$
$([3,0,3],\frac{1}{64}),([2,0,3],\frac{1}{64}),([1,0,3],\frac{1}{64}),([0,0,3],\frac{1}{64}),([3,1,3],\frac{1}{64}),([2,1,3],\frac{1}{64}),$
$([1,1,3],\frac{1}{64}),([0,1,3],\frac{1}{64}),([3,2,3],\frac{1}{64}),([2,2,3],\frac{1}{64}),([1,2,3],\frac{1}{64}),([0,2,3],\frac{1}{64}),$
$([3,3,3],\frac{1}{64}),([2,3,3],\frac{1}{64}),([1,3,3],\frac{1}{64}),([0,3,3],\frac{1}{64})]$

However these probabilities are not all that interesting, since they are all $\frac{1}{n^k}$. Most of the time we are actually more interested in the probabilities of certain events over the possible choices. We can do this with a function

```
eventOnkofn :: ((Either String [Integer], Rational)
                    -> (Either String Bool, Rational))
                -> Integer -> Integer -> [(Either String Bool, Rational)]
eventOnkofn event k n = groupDist $ map event $ choosekofn k n
```

that maps such an event over our computed possible choices and groups the result. One possible event could be "The same choice is made twice", specified through

```
eventDouble :: Eq a => (Either String [a], p) -> (Either String Bool, p)
eventDouble (e, p) = (fmap hasDuplicates e, p)
```

where `hasDuplicates` behaves as expected. Now we can call the function for the distribution we obtained above and get the more interesting result
$[(False,\frac{3}{8}),(True,\frac{5}{8})]$

While this specific computation executes in a timely manner, the amount of possibilities for a general ordered choice of k out of n with duplicates is n^k, meaning `choosekofn` has, for fixed k, polynomial time complexity in n, and for fixed n even exponential time complexity in k.
To circumvent this we can use Monte-Carlo-simulation, since calculating just one possibility has time complexity $\mathcal{O}(k \cdot n)$ and the Monte-Carlo-simulation is in turn linear in the number of repetitions, meaning we stay polynomial in any case. Even though this is a big time save, for large values calculation time can still become rather long, so we will only take a look at one small concrete example here. In the next section we will see a bigger application.
For this we define a function `choosekofnMC`, which in addition to k and n also takes the number of repetitions and uses `monteCarlo` instead of `reduceNeedDist`. We can also accordingly define `eventOnkofnMC`.

We will look at the case $k = 5, n = 10$ with our duplication event as laid out above. Even though these numbers are not very large, there are already $10^5 = 100000$ cases to compute. On top of that `fromnTuple` and `groupDist` both have a run time complexity of $\mathcal{O}(n^2)$ in that amount of possibilities. All of this results in `eventOnkofn 5 10` taking nearly 75 hours to compute with output:
$[(False, \frac{189}{625}), (True, \frac{436}{625})]$
However with `choosekofnMC` we can simulate this choice 10000 times within a reasonable 16 minutes and get the following distribution for the duplication event:
$[(False, \frac{2953}{10000}), (True, \frac{7047}{10000})]$
which is very close to the actual probability, being off by only 0.7%.

5.2 Probabilistic Tree Growth

This problem is taken from section 3 of [7], where it is used in context of probabilistic functional programming with monadic structures in Haskell.

We will model the life span of a tree, for which we have three possible states: alive at a certain height, hit by lightning at a certain height or toppled by a storm. In [7] a data type is used in order to encode these states, but since this calculus is untyped we will instead use Church encodings to represent the current height of the simulated tree. We will construct a recursive function that mimics this behaviour, each iteration either growing the tree by a random value between `one` and `five`, or killing it with either lightning or a storm, meaning the function stops with the current height or maps to `zero`.

First of all we need a function that returns a number from 1 to 5:

$$\texttt{prob1to5} = (\texttt{one} \oplus_{p_1} (\texttt{two} \oplus_{p_2} (\texttt{three} \oplus_{p_3} (\texttt{four} \oplus_{p_4} \texttt{five}))))$$

We can choose any probabilities we would like, here we will choose $p_1 = \frac{7}{100}, p_2 = \frac{8}{31}, p_3 = \frac{38}{69}, p_4 = \frac{24}{31}$ to get a probability distribution similar to a normal distribution, thus making extremely high/low values less likely. The resulting `ProbDistribution` is:

$$[(\texttt{five}, \frac{7}{100}), (\texttt{four}, \frac{6}{25}), (\texttt{three}, \frac{19}{50}), (\texttt{two}, \frac{6}{25}), (\texttt{one}, \frac{7}{100})]$$

Next we will need a function that performs the actual computation of one iteration of growth. This is accomplished by adding the current value to `prob1to5`, or choosing `end`, which will both be handled with the probability operator, where `end` is either the given variable or `zero`:

$$\texttt{end} = n \oplus_{p_e} \texttt{zero}$$

$$\texttt{probFunc} = \lambda f.\lambda n.((f(\texttt{plus prob1to5 } n)) \oplus_{p_t} \texttt{end})$$

Lastly we need to call this function recursively, where f will always be `probFunc` and n the current height. For the recursive call we will use the Y-combinator:

$$\texttt{ycomb} = \lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x))$$

We can once again choose p_e and p_t however we like, we will use $p_e = 0.7, p_t = 0.8$ here. Now we just need to start the recursion with initial height `zero` using `treeGrowth`:

$$\texttt{treeGrowth} = (\texttt{ycomb probFunc zero})$$

Now we are ready to reduce this function with `reduceNeed`. However as already seen in example 2.3.4, if we reduce Church numerals, the WHNF we get does not make it very easy to make out the actual numeral. This becomes even more prominent once we reach bigger numbers. One option to solve this is to apply a sort of post-processing in form of a call-by-value like reduction of the numerals.

This is possible in the interpreter through

```
reduceChurch :: ProbDistribution String -> ProbDistribution String
reduceChurch [] = []
reduceChurch ((Right e, p):xs) =
    case tryNeed e fresh True True Need 1 0 of
        PLeaf e' -> (Right e', p) : reduceChurch xs
        PNode _ (Left s) (Left s') -> (Left s, p) : reduceChurch xs
        PNode _ _ _ -> error "This is not a Church numeral"
reduceChurch (x:xs) = x : reduceChurch xs
```

which reduces the expressions in the given Distribution with the parameter `ch` in `tryNeed` set to `True` which, as explained in Section 4.2.1, basically extends reduction contexts from A- to C-contexts. Furthermore one can use `mapChurchToInt` to convert a `ProbDistribution` that contains only fully reduced Church numerals into a `[(Either String Integer, Rational)]` for even better readability. This solution is guaranteed to work, but has the drawback of getting computationally expensive for higher iterations of `treeGrowth`.
A less intrusive option, that however only returns the `Integer`'s for a given `treeGrowth` result and does not return the reduced numerals as expressions, is given in

```
mapChurchToIntNeed :: Eq a => ProbDistribution a
                           -> [(Either String Integer, Rational)]
```

which calls

```
churchToIntNeed :: Eq a => Expr a -> [Assignment a] -> Integer
churchToIntNeed (let env in e) old = churchToIntNeedEnv env env + 1
churchToIntNeed (\v.(\w.(\y.e))) _ = 0
churchToIntNeed (\v.(\w.e)) _ = countVar v e
churchToIntNeed (e1 e2) env =
    checkForSucc e1 env True + checkForSucc e2 env False
churchToIntNeed _ _ = 0
```

with

```
checkForSucc :: Eq a => Expr a -> [Assignment a] -> Bool -> Integer
checkForSucc (Var v) env True =
    case getVarChain v env env Need False of
        Just (\v.(\w.(\y.e))) -> 1
        _ -> 0
checkForSucc (e1 e2) env _ =
    checkForSucc e1 env True + checkForSucc e2 env False
checkForSucc _ _ _ = 0
```

This function adds all numbers in the environment, one for the successor function, which is what is in function position in the expression inside the `Letrec` after a call of `plus`, and then one for each successor that would be in the left side of an `Application`, since they would be applied to a number later on. For a reduced expression using only Church numerals and `plus` these are the only options after successful $\rightarrow$-reduction. This might however not work for other Church operations, hence for those either a different function must be written or `reduceChurch` must be used.

Now if we call `reduceNeedDist treeGrowth True Need`, the result will depend a lot on the threshold of steps specified in `tryNeed`. We have nested `Prob` statements in `probFunc`, where the outer one reduces to either 2 paths that terminate quickly, or to an expression that will have to evaluate `prob1to5`, which splits the current path into 5 paths. Since this is done in each and every recursive call, the number of possible paths extends greatly with progressive height of the `ProbTree`, which in turn is directly influenced by the number of reduction steps.
This means, the time complexity for `reduceNeedDist treeGrowth` is in $\mathcal{O}(5^n)$ with n being the amount of reduction steps. On the other hand, the lower the step threshold, the lower the possible maximum height, and the less accurate the probabilities for the computed values become, since for example a height of 13 could be achieved through growing 13 times by 1, growing 4 times by 3 and once by 1, etc. Many of these possibilities will vanish in the `Left` expression if the amount of steps is too low. With a 100 step threshold we get the following distribution after evaluating it with `mapChurchToIntNeed` and grouping probabilities for the same numbers:

$[(0, \frac{56190852299}{152587890625} \approx 36.825\%)$
$(1, \frac{49}{6250} \approx 0.784\%)$
$(2, \frac{21343}{781250} \approx 2.732\%)$
$(3, \frac{4452651}{97656250} \approx 4.560\%)$
$(4, \frac{1809598}{48828125} \approx 3.706\%)$
$(5, \frac{2779399}{97656250} \approx 2.846\%)$
$(6, \frac{1373421}{48828125} \approx 2.813\%)$
$(7, \frac{2738463}{97656250} \approx 2.804\%)$
$(8, \frac{46102}{1953125} \approx 2.360\%)$
$(9, \frac{922474}{48828125} \approx 1.889\%)$
$(10, \frac{55139}{3906250} \approx 1.412\%)$
$(11, \frac{848463}{97656250} \approx 0.869\%)$
$(12, \frac{194796}{48828125} \approx 0.399\%)$
$(13, \frac{61887}{48828125} \approx 0.127\%)$
$(14, \frac{12348}{48828125} \approx 0.025\%)$
$(15, \frac{2401}{97656250} \approx 0.002\%)$
$(\text{"Process terminated automatically after 100 steps"}, \frac{54697819576}{152587890625} \approx 35.847\%)]$

This calculation took around 37 seconds, but also around 35.8% of possible results are not computed.

If we extend the amount of steps by 50% we get this distribution:

$[(0, \frac{119923169273173831}{298023223876953125} \approx 40.240\%)$
$(1, \frac{49}{6250} \approx 0.784\%)$
$(2, \frac{21343}{781250} \approx 2.732\%)$
$(3, \frac{4452651}{97656250} \approx 4.560\%)$
$(4, \frac{452416307}{12207031250} \approx 3.706\%)$
$(5, \frac{21728519512}{762939453125} \approx 2.848\%)$
$(6, \frac{4311521809}{152587890625} \approx 2.826\%)$
$(7, \frac{8728446293}{305175781250} \approx 2.860\%)$
$(8, \frac{773714228}{30517578125} \approx 2.535\%)$
$(9, \frac{7030855069}{305175781250} \approx 2.304\%)$
$(10, \frac{33331093803}{1525878906250} \approx 2.184\%)$
$(11, \frac{1240197399}{61035156250} \approx 2.032\%)$
$(12, \frac{563939173}{30517578125} \approx 1.848\%)$
$(13, \frac{2535236256}{152587890625} \approx 1.661\%)$
$(14, \frac{2209635064}{152587890625} \approx 1.448\%)$
$(15, \frac{18097223221}{1525878906250} \approx 1.186\%)$
$(16, \frac{1361461514}{152587890625} \approx 0.892\%)$
$(17, \frac{920324181}{152587890625} \approx 0.603\%)$
$(18, \frac{376194364}{152587890625} \approx 0.247\%)$
$(19, \frac{73557036}{152587890625} \approx 0.048\%)$
$(20, \frac{2158499}{61035156250} \approx 0.004\%)$
$(21, \frac{117649}{305175781250} \approx 0.00004\%)$
$(\text{"Process terminated automatically after 150 steps"}, \frac{133826488663417963}{59604647753906250} \approx 22.452\%)]$

Calculating this distribution took around 2.5 hours, but here only around 22.5% of possible results are left uncomputed. All in all we can see that for larger simulations, calculating the full distribution is not practical or even feasible. However here we can observe, that the probability for continuing a path that yields a result larger than 11 is less than 1% with 100 steps and less than 8% with 150 steps. So by the nature of this function, the probability of paths that do not terminate tends to zero. This makes this a perfect example to use Monte-Carlo-simulation on.
While `reduceNeed` `treeGrowth` has exponential time complexity in the maximum number of steps, the maximum number of steps can be completely disregarded for `monteCarlo` `treeGrowth`. Furthermore the time complexity of Monte-Carlo-simulation is only $\mathcal{O}(n)$ in the number of repetitions computed.

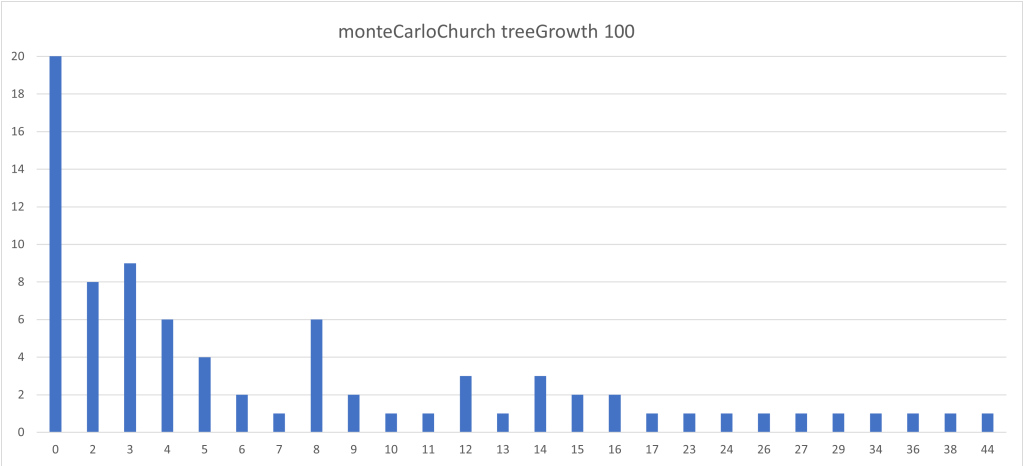

Figure 1: Plot for `monteCarloChurch treeGrowth 100`

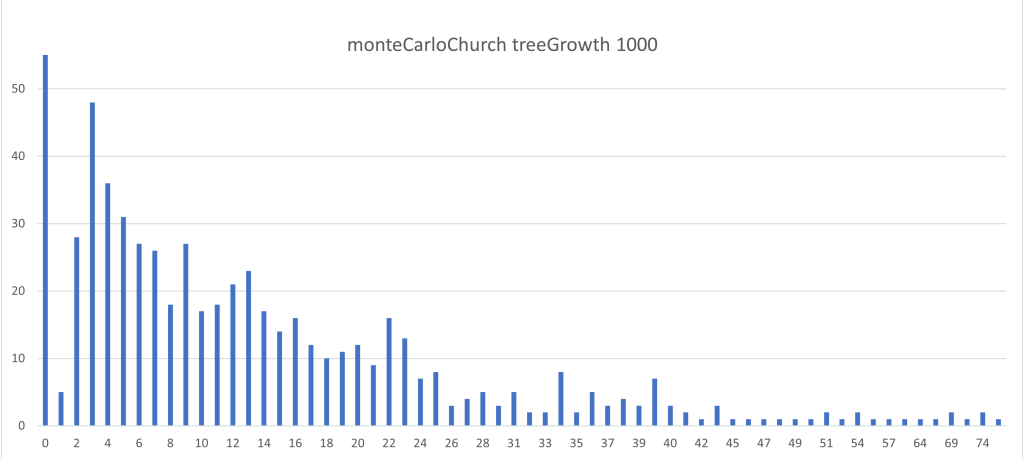

Figure 2: Plot for `monteCarloChurch treeGrowth 1000`

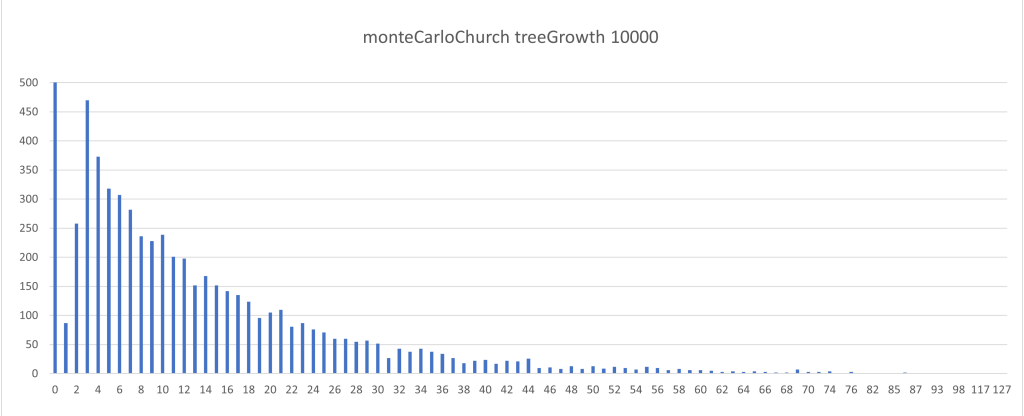

Figure 3: Plot for `monteCarloChurch treeGrowth 10000`

Fig. 1, Fig. 2 and Fig. 3 show that we get a pretty good normal distribution (with the number of occurrences on the y-axis and the respective values on the x-axis) with higher repetitions, while the computations took 12 seconds, 4.5 minutes and 45 minutes respectively. The graphs are zoomed in in order to make the differences between values apart from `zero` more apparent, since `zero` actually occurred around 10 times more often than the peak among the other values. They were computed using `monteCarloChurch` which is just a composition of `monteCarlo`, `mapChurchToIntNeed` and `groupDistCurch`, where the later groups by the `Integer`'s that are derived from the Church numerals.
In comparison to the full computation of the multi-distribution, this method produces results a lot quicker and in a much better scope. We also do not have to factor in possible left out paths, since we can reliably fully compute all repetitions, since stochastically speaking the possibility infinite calculation time is 0.

5.3 Comparing Strategies

Lastly, we will look at the two examples given above in regards to how different choices of `Strategy`, representing the variants of the $(\oplus)$-reduction in the interpreter, affect the efficiency. We will measure this through the number of reduction steps computed. We will also use garbage collection for all calculations in order to minimize computation time, since this does not affect the number of reduction steps taken.
For this purpose we will amend the `PLeaf` case of the `ProbTree` type to additionally store an `Integer`. In `tryNeed` we then store the counter we use to gauge when to forcefully terminate reduction, into every generated `PLeaf`. We can then define a function that counts the total number of steps:

```
countSteps :: ProbTree a -> Integer
countSteps (PLeaf _ ctr) = ctr
countSteps (PNode _ (Right (tl,_)) (Right (tr,_))) =
    countSteps tl + countSteps tr
countSteps (PNode _ (Right (tl,_)) _) = countSteps tl
countSteps (PNode _ _ (Right (tr,_))) = countSteps tr
countSteps _ = 0
```

For `choosekofn` we need to define an additional function for counting the total steps, since it uses reduction in both the function itself and in the call of `fromnTuple`.

```
countchoosekofn :: Integer -> Integer -> Strategy -> Integer
countchoosekofn k n s =
    let pool = makeProbExpr $ uniform $ take (fromIntegral n) allChurchNums
        reduce = testReduceStrategyGC (choosekofnRec k k pool) s
        count1 = countSteps reduce
        count2 = countfromTuple s k $ treeToDist reduce
    in count1 + count2
        where
            choosekofnRec k 1 pool =
                ((nTuple k) pool)
            choosekofnRec k ctr pool =
                ((choosekofnRec k (ctr-1) pool) pool)
```

The function `countfromTuple` simply counts the number of steps in the reductions made by `fromnTuple` with a specified `Strategy`.
Averaging the number of steps accross $k \in \{1,2,3\}, n \in \{4,5,6,7\}$ we get the following result:

$$\texttt{Need} : 3731, \quad \texttt{Name} : 3731, \quad \texttt{Value} : 1553533$$

We can see that `Name` and `Need` performed exactly the same, while `Value` performed far worse.
We see similar results when using Monte-Carlo-simulation, where we average the total steps across the number of repetitions, averaging those results over the values for k and n as above:

$$\texttt{Need} : 10, \quad \texttt{Name} : 10, \quad \texttt{Value} : 13$$

In this example all paths terminate, meaning reduction terminates regardless of `Strategy`. However the second example, probabilistic tree growth, has an infinite path. That leads to reduction not terminating in every case when using `Value`, as we have already seen in example 2.3.5. For `Need` and `Name` with the maximum number of steps set to 100, we get the following number of steps:

`Need` : 348404, `Name` : 349126

We see that `Name` performed 0.207% worse, taking 722 more steps. This difference however is negligible. For Monte-Carlo-simulation over 10000 repetitions, where computation with `Value` still does not terminate, we get similar results:

`Need` : 105, `Name` : 105

6 Conclusion

In this thesis, we have defined a probabilistic call-by-need λ-calculus, based upon L_{Need} of [3], for which we then explored the concept of contextual equivalence within it. On this topic we have compared different definitions of convergence, ultimately settling on probabilistic convergence, and defining expressions to be contextually equivalent if they converge with equal probability.
We have then seen an implementation of an interpreter for this calculus in Haskell. The implementation includes all reduction rules of Λ_{PNeed} as well as an implementation of Monte-Carlo-simulation, where a path is taken through random chance with the `System.Random` package.
Finally we saw two examples of probabilistic problems, represented them as expressions of Λ_{PNeed}, using first the Church encoding of pairs, which we expanded to n-tuples, and also the Church numerals. In addition to viewing the runtime-complexity of these examples, we also compared it to the runtime-complexity of Monte-Carlo-simulation and found that it was more efficient for higher values in both cases, while maintaining a sufficient approximation to the expected deterministic results.
In the end we furthermore compared the call-by-need strategy to the call-by-name and call-by-value strategies, in regards to evaluating $\oplus_p$-expressions, and found that call-by-need is at least as good as the other strategies in terms of runtime, with call-by-value being worse and call-by-name very similar.
Further work could consist of expanding the calculus and interpreter by types which would eradicate the need for Church encodings and the entailed call-by-value like reduction that is required to fully evaluate computations using Church numerals, making the calculus more easy to work with. One could also compare the interpreter against an implementation of a proper call-by-name probabilistic λ-calculus, such as the one given in [1], in regards to efficiency. Lastly one might verify confluence on this calculus, since [1] did not consider the call-by-need case specifically.

References

[1] Claudia Faggian and Simona Ronchi Della Rocca. Lambda calculus and probabilistic computation. In *34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, BC, Canada, June 24-27, 2019*, pages 1–13. IEEE, 2019.

[2] Ugo Dal Lago, Giulio Guerrieri, and Willem Heijltjes. Decomposing probabilistic lambda-calculi. In Jean Goubault-Larrecq and Barbara König, editors, *Foundations of Software Science and Computation Structures - 23rd International Conference, FOSSACS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*, volume 12077 of *Lecture Notes in Computer Science*, pages 136–156. Springer, 2020.

[3] David Sabel. Automating the diagram method to prove correctness of program transformations. *Electronic Proceedings in Theoretical Computer Science*, 289:17–33, Feb 2019.

[4] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 34(4):839–864, 1933.

[5] Manfred Schmidt-Schauss, David Sabel, and Elena Machkasova. Simulation in the Call-by-Need Lambda-Calculus with letrec. In Christopher Lynch, editor, *Proceedings of the 21st International Conference on Rewriting Techniques and Applications*, volume 6 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 295–310, Dagstuhl, Germany, 2010. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[6] David Sabel. Fortgeschrittene Funktionale Programmierung, Summer Semester 2020. URL: `https://www.tcs.ifi.lmu.de/lehre/ss-2020/fun/material/skript`.

[7] Martin Erwig and Steve Kollmansberger. Functional pearls: Probabilistic functional programming in Haskell. *J. Funct. Program.*, 16(1):21–34, 2006.