GOETHE UNIVERSITÄT FRANKFURT AM MAIN

www.uni-frankfurt.de

Einführung in die Funktionale Programmierung:

Funktionale Kernsprachen

Zufall Programmieren

Prof. Dr. Manfred Schmidt-Schauß

WS 2024/25

Stand der Folien: 6. Dezember 2024

Übersicht

1. Probability, functional
2. Verteilungen
3. Korrekte Programmtransformationen

Probability und Funktionale Programmierung

Ideen

- Einführung einer Funktion `coin` die einem Münzwurf entspricht
  in einer lazy funktionalen Programmiersprache
- Mit programmierbarer Wahrscheinlichkeit;
- in KFPTS: Zahlen und andere Datentypen vorhanden
- Rekursive Funktionen sind programmierbar

Probability und Funktionale Programmierung

Sichtweise: Was ist ein Programm?

- Ein Programm bzw. eine Funktion modelliert ein Zufalls-Experiment.
- Beispiele:
  - fairer Münzwurf
  - Würfeln mit einem Würfel mit 6 Seiten.
  - komplexeres Experiment: Programm mit mehreren Münzwürfe, Würfeln usw.
    Würfle solange bis ein 6 erscheint usw. . .

! verschiedene Ausführungen eines Programms haben verschiedene Ergebnisse.

Basis - Zufalls - Funktion

`coin` p s t

- p: Wahrscheinlichkeit für s
  $0 \leq p \leq 1$ rationale Konstante.
  (Man kann auch beliebige rationalwertige Ausdrücke zulassen)
- s, t sind die beiden möglichen Ausdrücke die als Fortsetzung gewählt werden können.
- Auswertung von `coin` p s t ist wenn $0 \leq p \leq 1$:
  - s mit Wahrscheinlichkeit p
  - t mit Wahrscheinlichkeit $1 - p$

  und dann weitere Auswertung.
  Wenn $p < 0$, dann wird $p = 0$ angenommen.
  Wenn $p > 0$, dann wird $p = 1$ angenommen.

`coin` gibt es nicht in Haskell!

Probabilistische Ausführung von `coin`

- Auswertung von `coin` p s t:
- Es wird **nicht-deterministisch** s oder t ausgewählt.
  D.h. Jede Ausführung kann mal s oder auch t wählen.
- Was ist mit der Wahrscheinlichkeit p?
- Wahrscheinlichkeiten spielen nur eine Rolle bei **mehrmaliger** Auswertung:
  der erste Ausdruck hat Wahrscheinlichkeit p, der zweite $1 - p$.

- Ein Programm P hat eine theoretische Verteilung der Ergebnisse (inkl. Nichtterminierung)
- Viele Ausführungen des Programms nähern diese Verteilung an. (Grenzwert-Satz)
- Eine Ausführung des Programm kann auch nicht-terminieren: D.h. die Verteilung zu P beinhaltet eine Wahrscheinlichkeit für Nichtterminierung.

KFPTSProb: Syntax mit: `coin`, `let` und Integer

$$
\begin{array}{lll}
\mathbf{Expr} & ::= & V \mid \lambda V.\mathbf{Expr} \mid (\mathbf{Expr}_1\ \mathbf{Expr}_2) \\
& \mid & (c_i\ \mathbf{Expr}_1\ \ldots\ \mathbf{Expr}_{\mathrm{ar}(c_i)}) \\
& \mid & (\mathsf{case}_{\mathsf{Typ}}\ \mathbf{Expr}\ \mathtt{of}\ \{\mathbf{Pat}_1 \to \mathbf{Expr}_1; \ldots; \mathbf{Pat}_n \to \mathbf{Expr}_n\}) \\
& \mid & SK \text{ wobei } SK \in \mathcal{SK} \\
& \mid & (\mathit{let}\ V_1 = \mathbf{Expr}_1, \ldots, V_n = \mathbf{Expr}_n\ \mathtt{in}\ \mathbf{Expr}) \\
& \mid & 0 \mid 1 \mid 2 \ldots \mid + \mid - \mid * \\
& \mid & (\mathtt{coin}\ \mathbf{Q}\ \mathbf{Expr}_1\ \mathbf{Expr}_2) \\
\mathbf{Pat}_i & ::= & (c_i\ V_1 \ldots V_{ar(c_i)}) \text{ wobei die Variablen } V_i \text{ alle verschieden sind.} \\
\mathbf{Q} & ::= & \text{rationale Zahl}
\end{array}
$$

Pro Superkombinator SK gibt es eine Superkombinatordefinition:
$SK\ V_1\ \ldots\ V_n = \mathbf{Expr}$ mit $n = ar(SK)$
`coin` $\mathbf{Q} \ldots$: allgemeiner: $\mathbf{Q}$ Expression der zur rational auswertet.

Probabilistisches Programm

Beispiel:

```
main = a + b
a    = coin 0.1 10 20
b    = coin 0.25 3 4
```

Programm

- Ist ein Modell für ein Zufalls-Experiment
- Man kann diese Experimente kombinieren (pogrammieren)
- Man kann das Programm optimieren bzw. transformieren in ein äquivalentes Programm
- Programm ausführen entspricht einem Experiment
- Programme kombinieren: komplexere Experimente.

Probabilistische call-by-need Auswertung

Auswertung ist **call-by-need** in `KFPTSProb`.
(Normalordnungs-Reduktion mit Sharing)
Eine detaillierte exakte Definition siehe Literatur zu n.d. FP-calc.

Call-by-need Details und Prinzipien

1. Der Normalordnungs-Redex wird zuerst bestimmt.
2. Das `let` ist normalerweise mit mehreren Bindungen und rekursiv
3. Sharing wird modelliert durch let-Referenzen.
4. Da rekursives let; und Sharing zu beachten sind: genaue Reduktionsregeln sind leider einige und kompliziert

Probabilistische call-by-need Auswertung

Exkurs: call-by-need Auswertung

1. let-Ausdrücke im Fokus (d.h. Normalordnungs-Kontext) werden
   "nach oben" geschoben wenn nötig.
   vereinigt usw.
2. Bei LBeta und Case-Reduktion wird Einsetzung geshared (mittels let)
3. Beta Reduktion ist mit sharing:
   $(\lambda x.s)\ t \quad \to \quad \texttt{let}\ x = t\ \texttt{in}\ s$
4. Echt kopiert (d.h. verdoppelt) werden dürfen nur Abstraktionen $\lambda x.s$ und einfachste Konstruktorapplikationen der Form $c\ x_1 \ldots, x_n$
5. Man darf zwei textuelle getrennte Applikationen und `coin`-Aufrufe nicht mittels einer Programmtransformation "sharen".

Probabilistische call-by-need Auswertung

Einige Reduktionsregeln:

$(\lambda x.s)\ t \quad \xrightarrow{lbeta} \quad \texttt{let}\ x = t\ \texttt{in}\ s$

$(\texttt{case}\ (c\ s_1\ s_2)\ \texttt{of}\ (c\ x_1\ x_2) \to t; \ldots) \quad \xrightarrow{lcase} \quad \texttt{let}\ x_1 = s_1; x_2 = s_2\ \texttt{in}\ t$

$((\texttt{let}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ s)\ t) \xrightarrow{let} \quad (\texttt{let}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ \ (s\ t))$

$((\texttt{let}\ x_1 = (\texttt{let}\ y_1 = r_1, \ldots, y_m = r_n\ \texttt{in}\ s_1), x_2 = s_2, \ldots, x_n = s_n\ \texttt{in}\ r))$
$\xrightarrow{let} \quad (\texttt{let}\ y_1 = r_1, \ldots, y_m = r_n, \quad x_1 = s_1, x_2 = s_2, \ldots, x_n = s_n\ \texttt{in}\ \ r$

• Es gibt noch weitere Regeln um let-Bindungen nach oben zu verschieben

Probabilistische call-by-need Auswertung, Beispiel

```
let y = coin 0.75 1 2; z = coin 0.25 3 4 in
   let x = coin 0.5 y z in x*y+z
```

$\longrightarrow$

```
let y = coin 0.75 1 2; z = coin 0.25 3 4,
   x = coin 0.5 y z in x*y+z
```

$\longrightarrow x$ zuerst; Wahl: wird zu y. $p = 0.5$

```
let y = coin 0.75 1 2; z = coin 0.25 3 4,
  x = y in x*y+z
```

$\longrightarrow y$ auswerten, $p = 0.5 * 0.25$

```
let y = 2; z = coin 0.25 3 4,
  x = y in x*y+z
```

$\longrightarrow z$ auswerten, $p = 0.5 * 0.25 * 0.25$

```
let y = 2; z = 3,
  x = y in x*y+z
```

$\longrightarrow 7$, $p = 0.5 * 0.25 * 0.25 = 1/32$

Prob cb-need Auswertung, Beispiel Forts.

Anmerkungen

- Es kommen Bindungen $x = y$ vor:
  $\Longrightarrow$ Kalkülregeln verfeinern, damit das abgedeckt ist.
- Es gibt 8 mögliche Ausführungen. Wahrscheinlichkeiten:
  $\{0.25, 0.75\} * \{0.25, 0.75\} * \{0.5, 0.5\}$
- Nur eine Möglichkeit ist auf der Folie.
- Auswertung hängt vom Ausdruck $x * y + z$ ab.
- Auswertung hängt auch vom Zufall ab:
  Wenn Ausdruck $= x$, dann wird y oder z ausgewertet.

Monte Carlo Simulation; Beispiel

- Liste der Länge n:
  `list = [coin 0.5 0 1,...,coin 0.5 0 1]`
- Berechne `m = sum list`. (normalerweise in etwa $n/2$).
- Wenn n sehr groß ist wird $|m/n - 0.5|$ immer kleiner.

Technisches Problem, wenn man es in `KFPTSProb` macht:

Obige Programmiertechnik verlangt,
dass man n mal `coin 0.5 0 1` im Programm hinschreibt!

Monte Carlo Simulation: Programmierbar

Gegeben; geschlossener Ausdruck s in `KFPTSProb`.

1. Werte {einen Ausdruck } 1000 mal aus.
2. Mache Statistik über alle Ergebnisse und Wahrscheinlichkeiten

Simulationen mit paramtrisierter Anzahl.

- n Münzwürfe sollen gemacht werden:
- `listCoins = repeat n (\x. (coin 0.5 0 1))`
  Trick: Abstraktionen zu kopieren statt zu sharen ist korrekt
- Berechne Summe aller Ergebnisse:
  `sum (map (\x-> x ()) listCoins)`
- Auswertung kopiert die Abstraktion `(\x. (coin 0.5 0 1))`
- Programm enthält nur einen Ausdruck `(coin 0.5 0 1)`!

Grenzwertsatz: Je mehr Versuche, desto größer ist die Wahrscheinlichkeit, dass der Mittelwert sehr nah bei 0.5 ist.

Probabilistischer Berechnungsbaum

Sei P ein `KFPTSProb` Programm.
Berechnung aller Möglichkeiten erzeugt einen Baum!

- Wurzelknoten ist das Programm
- Die Kanten sind die normal-order Reduktionen, markiert mit den Wahrscheinlichkeiten
- Wenn `(coin p 0 1)` ausgewertet wird, hat der Knoten zwei Kinder.
- Die Blätter sind WHNFs.

Probabilistischer Baum zu einem Programm P hat Möglichkeiten:

- kann endlich sein
- kann unendlich groß sein
- kann unendliche lange Äste haben

Die **Wahrscheinlichkeit eines Astes**:
ist das Produkt aller Wahrscheinlichkeiten an seinen Kanten.

Multi-Verteilung von P

Sei P ein `KFPTSProb` Programm.

Definition: Die Multi-Verteilung zu P ist

Menge der Paare (s, p) zu allen Ästen.

- s ist eine WHNF am Ende eines Astes,
- p die Wahrscheinlichkeit des Astes

Terminierungs-Wahrscheinlichkeit $=$
Summe aller Wahrscheinlichkeiten in der Verteilung.
Terminierungs-Wahrscheinlichkeit $EC(P)$ von P:
$=$ Summe aller Wahrscheinlichkeiten aller endlichen Äste
Nichtterminierungs-Wahrscheinlichkeit von P:
Summe aller Wahrscheinlichkeiten aller unendlichen Äste

Simulationsprogramm dazu mit Multi-Verteilungen sind einfacher als mit Verteilungen
(insbesondere bei unendlichen Multi-Verteilungen.)

Nichtterminierung und Wahrscheinlichkeit

Gut zu wissen: Es gibt programmierte Zufallsexperimente,
deren (rekursive) Ausführung
mit positiver Wahrscheinlichkeit
nicht terminiert
und die keine programmierte Endlos-Schleife haben

(Ein Beispiel kommt noch)

Verteilung von P

Sei P ein `KFPTSProb` Programm.

Berechnung der Verteilung

Eine **Verteilung** $EV_P(.)$ zum Programm P : `int`

- $EV_P(n) =$ Wahrscheinlichkeit, dass P die Zahl n berechnet.
- $EV_P(\bot)$: Wahrscheinlichkeit der Nichtterminierung von P

Es gilt: $(\sum_i EV_P(i)) + V_P(\bot) = 1$

Beispiel: Würfel

Fairer 6er Würfel

$$\texttt{wuerfel} = \texttt{coin}\ (1/6)\ 1\ (\texttt{coin}\ (1/5)\ 2\ (\texttt{coin}\ (1/4)\ 3\ (\texttt{coin}\ (1/3)\ 4\ (\texttt{coin}\ (1/2)\ 5\ 6)))))$$

Begründung dass das richtg ist:

- Münzwurf: Prob $1/6$ für 1 und $5/6$ der Rest.
- Dann: Prob $5/6 * 1/5$ für 2: d.h.
  $1/6$ für 2 und $4/6$ für den Rest.
- usw.

$EV_P(\texttt{wuerfel}) = \{i \mapsto 1/6 \mid i = 1, \ldots, 6\}$

Verallgemeinerung: Zufälliges Element aus Liste

Faires Ziehen eines Listenelements aus `l`

```
gVert l     =  if (length l) == 0 then bot else
            =     gV l (length l)
gV [x] 1    =  x
gV (x:r) k  =  coin (1/k) x (gV r (k-1))
```


Beispiel: Zufallsprogramm zu diskreter Verteilung

Diskrete Verteilung, Programm verallgemeinert

$[(w_1, p_1), \ldots (w_n, p_n)]$

$$\texttt{wuerfel} = \texttt{coin}\ (p_1)\ w_1\ (\texttt{coin}\ (p_2/(1-p_1))\ w_2\ (\texttt{coin}\ (p_3/(1-p_1-p_2))\ w_2\ \ldots$$

Erwartungswert

Definition

Falls alle Ergebnisse Integer sind:
Erwartungswert $= \sum_i p_i * w_i$,
wenn $[(w_i, p_i), i = 1 \ldots, n]$ die Verteilung für Integer i ist.

Erwartungswert beim 6-Würfel ist $\sum_i i * p_i$, wobei
$[(1, 1/6), (2, 1/6), \ldots, (6, 1/6)]$
die Verteilung für den 6-Würfel ist.
Der Erwartungswert ist dann $1/6 * (\sum_{i=1}^{n}\ i) = 3.5$.

Vergleich der Auswertungsreihenfolgen

Verhalten bei beta-Reduktion:

- Call-by-value: Vor Reduktion muss das Argument ausgewertet werden.
- Call-by-name: Die Argumente werden bei beta-Reduktion in den Rumpf kopiert.
- Call-by-need: Die Argumente werden bei beta-Reduktion in den Rumpf kopiert, aber sind "geshared".

Resultate	call-by-value	call-by-need	call-by-name
$(\lambda x.1)\ \bot$	terminiert nicht	1	1
let w = coin 1 2 in w+ w	2,4	2,4	2,3,4

Wir benutzen in **KFPTSProb**:
Call-by-need Auswertung, wie in Haskell

Ein Beispiel mit unendlich vielen Werten

Erzeuge Werte 1,2,3,4,...
mit den Wahrscheinlichkeiten 0.5, 0.25, 2^{-3}, 2^{-4},....

$$\begin{array}{lcl} \texttt{result} & = & \texttt{numbers}\ 1 \\ \texttt{numbers}\ n & = & \texttt{coin}\ 0.5\ n\ (\texttt{numbers}\ (n+1)) \end{array}$$

- Man erhält $EC(\texttt{result}) = 1$, d.h. Terminierung mit Wahrscheinlichkeit 1.
- Die Verteilung $EV(\texttt{result})$ ist: $\lambda i.2^{-i}$.
- Das Programm kann unendlich lange laufen, wenn $coin$ immer falsch fällt, aber die Wahrscheinlichkeit dafür ist 0.

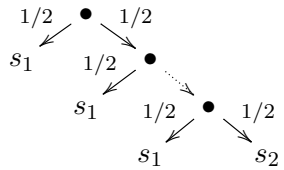
Nichtterminierung mit Wahrscheinlichkeit > 0

Beispiel-Programm das mit positiver Wahrscheinlichkeit nicht terminiert. (K ist eine Abstraktion $\lambda x.\lambda y.x$.)

$$\begin{array}{l} s := \texttt{let}\ cprob\ i\ x\ y = if\ i = 0\ then\ x\ else \\ \qquad\qquad\qquad\qquad coin\ 0.5\ (cprob\ (i{-}1)\ x\ y)\ \ y, \\ \qquad\quad gen\ i = cprob\ i\ K\ (gen\ (i{+}1)) \\ \quad \texttt{in}\ gen\ 2 \end{array}$$

Die Graphiken zeigen die Verzweigung

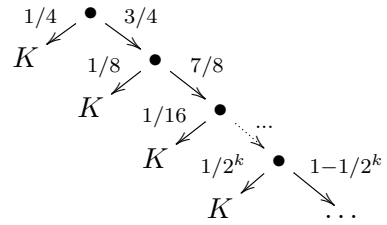
$cprob\ i\ s_1\ s_2$:



D.h., $cprob\ i\ s_1\ s_2$ wird mit Wahrscheinlichkeit $1/2^i$ zu s_2 und mit Wahrscheinlichkeit $(1 - 1/2^i)$ zu s_1.

Nichtterminierung mit prob > 0, Forts.

$gen\ 2$:



Der Aufruf $(gen\ 2)$ wird mit Wahrscheinlichkeit $1/4$ zu K
und mit Wahrscheinlichkeit $3/4$ geht es dann mit $(gen\ 3)$ weiter.

Grobe Abschätzung:
Terminierung von (gen 2)
mit Wahrscheinlichkeit $< 1/4 + 1/8 + \ldots = 1/2$.

Nichtterminierung mit prob > 0, Forts.

- Exakt: (gen 2) terminiert mit Wahrscheinlichkeit $5/12$, also: (gen 2) terminiert nicht mit Wahrscheinlichkeit $7/12$, d.h. $> 50\%$.
- $\Rightarrow$ Es gibt Programme, die mit $W > 0$ nicht terminieren,
  - ohne eine direkte Schleife, nur durch den rekursiven Ablauf!
  - Und jeder rekursive Aufruf hat positive Erfolgswahrscheinlichkeit

Vergleich mit nicht-deterministischen FPS:

Die ND-Terminierungs-Begriffe (ohne Probability)

- may-convergent: Der Ausdruck hat eine Möglichkeit mit WHNF zu terminieren.
- may-divergent: Der Ausdruck hat eine Möglichkeit zu divergieren (d.h. unendlich oder jedes Ende ist keine WHNF)
- must-convergent: Jede Reduktionsfolge endet mit einer WHNF.
- must-divergent: keine Reduktionsfolge endet mit einer WHNF.
- should-convergent: Jeder Reduktionsnachfolger ist may-convergent.

Unterschied ND vs. Probabilistisch

- Es gibt should-konvergente geschlossene Ausdrücke, die mit mehr als 0.5 Wahrscheinlichkeit nicht terminieren, (Aber < 1).
- Es gibt may-divergente Ausdrücke, die mit Wahrscheinlichleit 1 konvergieren.

Programmtransformationen in KFPTSProb

Was bedeutet **gleiches Programm** in KFPTSProb ?

Dazu:	Genaue Festlegung der Sprache notwendig! Und eine genaue Festlegung der Auswertung (genauer: der operationalen Semantik)
Begriff:	semantisch gleiche Programme bzw. semantisch gleiche Unter-programme

Programmtransformationen in KFPTSProb

Basisbegriff: *EC*: (expected convergence)
Erwartungswert für Terminierung (bzw. Konvergenz)

Definition

Für KFPTSProb-Ausdrücke s, t :
$s \sim_{c,P} t$ wenn für alle Kontexte C: $EC(C[s]) = EC(C[t])$,
d.h. wenn die Konvergenzwahrscheinlichkeit sich nicht ändert, wenn man s durch t ersetzt oder umgekehrt. Mit Kontext C sind KFPTSProb-Programme gemeint, die eine Leerstelle haben.

- Man kann ziemlich viele Programm-Äquivalenzen und -Transformationen in KFPTSProb als korrekt nachweisen.
- d.h. man kann Programme umformen, weiter auswerten usw., mittels korrekter Transformationen!
- Natürlich darf man bei Transformationen (z.B. im Compiler)) nicht die Münzwürfe ausführen.

Verteilungsäquivalenz in KFPTSProb

Definition

Zwei offene Programme P, Q mit main' sind äquivalent, wenn für jedes Programm R die Konkatenation $P|R$ (mit main) und $Q|R$ die gleiche Wahrscheinlichkeit der Konvergenz haben.

Welche Programmtransformationen sind korrekt?

Man kann zeigen, dass in KFPTSProb folgende korrekt sind:

- LBeta-Reduktion
- LCase-Reduktion

Vertauschung der Ausdrücke in coin Ausdrücken ist korrekt!
(auch wenn aktuell kein expliziter Beweis aufgeschrieben ist.)

Verteilungsäquivalenz in KFPTSProb

Sei KFPTSProb^{mt} monomorph getypte Variante von KFPTSProb dann gilt:
• Für Programme vom Typ Int: Kontextuelle Äquivalenz in KFPTSProb^{mt} ist äquivalent zu Verteilungsäquivalenz.

Das bedeutet: Wenn die Sprache einfach getypt ist,
z.B. $+$: int $\to$ int $\to$ int
Dann sind geschlossene Ausdrücke p_1 und p_2 (mit gleichem Typ und int als Ergebnistyp)
kontextuell gleich
gdw. beide die gleiche Verteilung auf $0, 1, 2, \ldots$ haben.

Fazit und Ausblick

Welche weiteren Vorteile hat das lazy (call-by-need) probabilistic Programmieren?

- Ein Programm ist ein programmiertes Zufallsexperiment.
- Die Programme sind unmittelbar geeignet zur zufälligen Auswertung mittels Monte-Carlo Methode.
- Man kann in nicht zu komplizierten Fällen deren Konvergenz-Wahrscheinlichkeit oder die Verteilung direkt bestimmen, ohne Monte-Carlo Methoden zu verwenden.
- Es gibt eine umfangreiche Forschung und Literatur zur probabilistischen Programmierung, wobei es zu lazy funktionalen Programmiersprachen noch nicht soviele Untersuchungen gibt.

Haskell Bibliothek probability

Die Haskell Bibliothek .../packages/probability hat den Ansatz, aus gegebenen diskreten Verteilungen weitere diskrete Verteilungen zu Zufallsprozessen zu berechnen.
Das hat Vor- und Nachteile:

- Man kann weitere Zufallsprozesse zusammensetzen und berechnet direkt deren Verteilung. z.B. die Verteilung der Ergebnisse wenn man zwei Würfel wirft.
- Es gibt weitere (auch direkt programmierbare) Kombinationsmöglichkeiten von Zufallsvariablen, bzw. deren Verteilungen.
- Diese Sichtweise ist etwas anders als die direkte (rekursive) Programmierung von Zufallsexperimenten. Es ist damit leichter, Verteilungen zu berechnen, aber es ist vermutlich nicht so allgemein wie die direkte Programmierung.

Haskell Bibliothek probability

Warum Multiverteilungen?

- Die Berechnung von Verteilungen liefert erstmal eine "Multi-Verteilung", d.h. zu einem x können mehrere (evtl. unendlich viele) Einträge $(x, p_1), (x, p_2)$ in der "Verteilung" sein.
- Zum Beispiel die unendliche Verteilung V:
  $[(1, 0.5), (2, 1/4), (3, 1/8), \ldots]$
  liefert für das Produkt von zwei Zahlen:
  1*1 mit $1/4$, aber für 2 als Ergebnis zwei Einträge:
  $(1 * 2, 1/8)$ und $(2 * 1, 1/8)$ zusammen $(2, 1/4)$.
- Multiverteilungen sind einfacher zu handhaben beim Programmieren

Literatur-Auswahl (Vorsicht..)

- (Übersichtsartikel) Ugo Dal Lago. 2020. On Probabilistic Lambda-Calculi. Cambridge University Press, 121-144. `https://doi.org/10.1017/9781108770750.005`
- (Artikel zum Anwendungspotential) Goodman, N. D., Tenenbaum, J. B., & Gerstenberg, T. (2014). Concepts in a probabilistic language of thought. Center for Brains, Minds and Machines (CBMM).
- (Konferenzartikel zu call-by-need probabilistic programs) D. Sabel, M. Schmidt-Schauß, and L. Maio. 2022. Contextual Equivalence in a Probabilistic Call-by-Need Lambda-Calculus. In 24th PPDP 2022, Tbilisi, Georgia. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3551357.3551374
- (Implementierung in Haskell) Luca Maio, The Probabilistic Lambda Calculus with Call-by-Need-Evaluation, thesis, LMU München, 2021
- Martin Erwig, Steve Kollmannsberger, J. Funct. Program. 16(1): 21–34 (2006) `web.engr.oregonstate.edu/(tilde)erwig/papers/PFP_JFP06.pdf`

Literatur-Auswahl 2.

- Artikel zur Programmäquivalenz: in einer probabilistischen, getypten funktionalen Programmiersprache, analog zu `KFPTSprob` (2023) M. Schmidt-Schauß, D. Sabel, Program equivalence in a typed probabilistic call-by-need functional language, J. Log. Algebraic Methods Program. 135, 2023, `https://doi.org/10.1016/j.jlamp.2023.100904`

Bibliotheken und Software

- Haskell probability: https://hackage.haskell.org/packages/probability
- Die Webseite zum Haskell Code der functional pearl zu probability von Martin Erwig, Steve Kollmannsberger: https://web.engr.oregonstate.edu/{tilde}erwig/pfp/