

Einführung in die Funktionale Programmierung:

Funktionale Kernsprachen Zufall Programmieren

Prof. Dr. Manfred Schmidt-Schauß

WS 2022/23

Stand der Folien: 22. November 2022

Ideen

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

Übersicht

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

Würfel - 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` ergibt
 - s mit Wahrscheinlichkeit p
 - t mit Wahrscheinlichkeit $1 - p$

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.

Probabilistisches Programm

KFPTSProb ist die Sprache KFPTSProb: das ist KFPTS + let + coin.

Auswertung ist **call-by-need** s.u.:

Programm

- Ist ein Modell für ein Zufalls-Experiment
- Man kann diese Experimente kombinieren (programmieren)
- 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.

Das ist Normalordnung mit Sharing.

Eine detaillierte exakte Definition siehe Literatur zu n.d. FP-calc.

Call-by-need Details und Prinzipien

- 1 Der Normalordnungs-Redex wird zuers bestimmt.
- 2 Das let ist normalerweise mit mehreren Bindungen. Es kann rekursiv sein oder nicht-rekursiv je nach Sprache.
- 3 Wenn nicht rekursiv, dann extra Fixpunktkombinator (und weitere Beschränkungen)
- 4 Wenn rekursives let, dann komplexere Reduktionsregeln
- 5 let-Ausdrücke werden nach oben geschoben wenn nötig.
- 6 Bei LBeta und Case-Reduktion wird Einsetzung geshared.
- 7 Sharing wird erreicht durch let-Referenzen.
- 8 Echt kopiert werden dürfen nur Abstraktionen und Konstruktorapplikationen der Form $c\ x_1 \dots x_n$

Probabilistische call-by-need Auswertung

Einige Reduktionsregeln:

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

$$(\text{case } (c\ s_1\ s_2) \text{ of } (c\ x_1\ x_2) \rightarrow t; \dots) \xrightarrow{lcase} \text{let } x_1 = s_1; x_2 = s_2 \text{ in } t$$

$$((\text{let } x_1 = s_1, \dots, x_n = s_n \text{ in } s) t) \xrightarrow{let} (\text{let } x_1 = s_1, \dots, x_n = s_n \text{ in } (s\ t))$$

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

- 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
```

→

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

→ 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
```

→ y auswerten, $p = 0.5 * 0.25$

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

→ z auswerten, $p = 0.5 * 0.25 * 0.25$

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

→ 7, $p = 0.5 * 0.25 * 0.25 = 1/32$

Prob cb-need Auswertung, Beispiel Forts.

Anmerkungen

- Es kommen Bindungen $x = y$ vor:
⇒ Kalkülregeln verfeinern, damit das abgedeckt ist.
- Es gibt 8 mögliche Ausführungen. Wahrscheinlichkeiten:
 $\{0.25, 0.75\} * \{0.25, 0.75\} * 0.5$
- 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

eines Programms bzw. Ausdrucks in KFPTSProb:

- 1 Werte einen Ausdruck mehrfach aus.
- 2 Mache Statistik über alle Ergebnisse

Im Fall von `coin 0.5 0 1`:

Ergebnisliste ist eine Liste mit Einträgen $\in \{0, 1\}$.

Der Mittelwert der Ergebnisse ist in der Nähe von 0.5

Grenzwertsatz: Je mehr Versuche, desto näher ist der Mittelwert an 0.5.

Monte-Carlo Simulation von `coin p 0 1`:
Mittelwert der Ausgänge geht gegen p .

Probabilistisches 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**:

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

Multi-Verteilung eines Programms P

Folgerung: Es gibt programmierte Zufallsexperimente, deren (rekursive) Ausführung mit **positiver** Wahrscheinlichkeit **nicht terminiert**.

Verteilung von P

Sei P ein KFPTsprob Programm.

Definition

Eine **Verteilung** $EV(P)$ zu P kann man berechnen, wenn man die (Un-)Gleichheit aller WHNFS an den Enden des Berechnungsbaumes entscheiden kann.

Z.B.

Wenn man nur ganze (oder natürliche) Zahlen aus Ausgänge hat.

Beispiel: Würfel

Fairer 6er Würfel

```
wuerfel = coin (1/6) 1 (coin (1/5) 2 (coin (1/4) 3
              (coin (1/3) 4 (coin (1/2) 5 6))))
```

Begründung

- 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 $4/6$ sonst.
- usw.

Erwartungswert

Definition

Falls alle Ergebnisse Integer sind:

Erwartungswert = $\sum_i p_i * w_i$, wenn (w_i, p_i) die Verteilung für integer i ist.

Erwartungswert beim 6-Würfel ist $\sum_i i * p_i$, wobei

$(1, 1/6), (2, 1/6), \dots, (6, 1/6)$

die Verteilung für den 6-Würfel ist.

Der Erwartungswert ist dann $1/6 * \sum_{i=1}^n i = 3.5$.

Ein Beispiel mit unendlich vielen Werten

Erzeuge Werte 1,2,3,4,...

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

```
result = numbers 1
```

```
numbers n = coin 0.5 n (numbers (n + 1))
```

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

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
$(\lambda x.1) \perp$	terminiert nicht	1	1
let w = coin 1 2 in w+ w	2,4	2,4	2,3,4

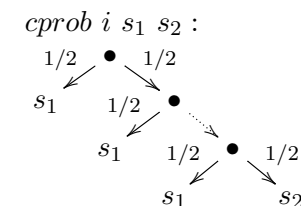
Wir benutzen in KFPSprob: Call-by-need Auswertung, wie es Haskell implementiert hat.

Nichtterminierung mit prob > 0

Beispiel-Programm das mit positiver Wahrscheinlichkeit nicht terminiert.

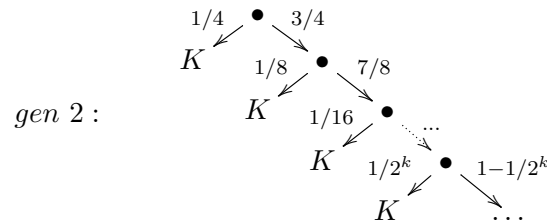
```
s := let cprob i x y = if i = 0 then x else
                        coin 0.5 (cprob (i-1) x y) y,
      gen i = cprob i K (gen (i+1))
in gen 2
```

Die Graphiken zeigen die Verzweigung



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 $\text{prob} > 0$, Forts.



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 kleiner als $1/4 + 1/8 + \dots = 1/2$.

Nichtterminierung mit $\text{prob} > 0$, Forts.

- Exakt: (*gen 2*) terminiert mit Wahrscheinlichkeit $5/12$
- *gen 2* terminiert nicht mit Wahrscheinlichkeit $7/12$, d.h. $> 50\%$.
- $\Rightarrow$ Es gibt Programme, die mit $W > 0$ nicht terminieren, nur durch den rekursiven Ablauf!

Vergleich mit nicht-deterministischen FPS:

Die ND-Terminierungs-Begriffen (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 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 50% Wahrscheinlichkeit nicht terminieren
- Es gibt may-divergente Ausdrücke, die mit Wahrscheinlichkeit 1 konvergieren.

Programmtransformationen in KFPTSProb

Was bedeutet **gleiches Programm** in KFPTSProb ?

Dazu: Genaue Beschreibung der Sprache notwendig!

- Man kann die Kernsprache KFPTS von Haskell nehmen, mit lambda, case, Konstruktoren usw. und call-by-need Auswertung.
- kann getypt oder auch ungetypt sein.
- Rekursion kann man mit dem `let(rec)` erreichen bzw. mit der rekursiven Definition der Superkombinatoren.
- Zusätzlich gibt es den Münzwurf (`coin p s t`), wobei p rationaler Ausdruck ist.
Bei $p \leq 0$ ist es wie Wahrscheinlichkeit 0, und bei $p \geq 1$ wie Wahrscheinlichkeit 1.

Programmtransformationen in KFPTSProb

Definition

$s \sim_{c,P} t$ 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 Programmtransformationen als korrekt nachweisen.
- d.h. man kann Programme umformen, weiter auswerten usw., mittels korrekter Transformationen!
- Natürlich darf man intern (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 einer Variante von KFPTSProb folgende korrekt sind:

- LBeta-Reduktion
- LCase-Reduktion

Vertauschung der Ausdrücke in `coin` Ausdrücken ist vermutlich auch korrekt

Verteilungsäquivalenz in KFPTSProb

Definition

Zwei geschlossene (getypte) Programme P_1, P_2 sind **Verteilungs-äquivalent**, $P_1 \sim_V P_2$ wenn sie die gleiche Verteilung der Werte erzeugen.

Unter weiteren Voraussetzungen an die Programmiersprache gilt (vermutlich):

- Kontextuelle Äquivalenz ist äquivalent zu Verteilungsäquivalenz.

Fazit und Ausblick

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

- 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 so viele Untersuchungen gibt.

Haskell Bibliothek probability

Die Haskell Bibliothek `.../packages/probability` hat den Ansatz, aus gegebenen diskreten Verteilungen weitere diskreten 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.

Literatur-Auswahl

- (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](http://web.engr.oregonstate.edu/~(tilde)erwig/papers/PFP_JFP06.pdf)

Bibliotheken und Software

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