Funktionale Programmierung

Sommersemester 2025

Prof. Dr. Jakob Rehof
M.Sc. Felix Laarmann
TU Dortmund
LS XIV Software Engineering

Diese Vorlesung:

- Kurze Wiederholung: Church Numerale
- Datenstrukturen im (ungetypten)λ-Kalkül
- „Programmieren" im (ungetypten) λ-Kalkül
- Haskell-Datenstrukturen ins (ungetypte) λ-Kalkül übersetzen

Church Numerale

2.14. Definition. (i) $F^n(M)$ with $F \in \Lambda$ and $n \in \mathbb{N}$ is defined inductively as follows.

$$\begin{aligned} F^0(M) &\equiv M; \\ F^{n+1}(M) &\equiv F(F^n(M)). \end{aligned}$$

(ii) The *Church numerals* $\mathbf{c}_0, \mathbf{c}_1, \mathbf{c}_2, \ldots$ are defined by

$$\mathbf{c}_n \equiv \lambda f x. f^n(x).$$

Church Numerale

- Die Definition der Church Numerale $c_n = \lambda f\ x\ .\ f^n(x)$ haben wir so ähnlich doch schon auf Übungsblatt 5 gesehen! Wir erinnern uns:

```
data Nat = Z | S Nat
```

```
nullN :: Nat
nullN = Z
```

$c_0 = \lambda f\ x\ .\ x$

```
einsN :: Nat
einsN = S Z
```

$c_1 = \lambda f\ x\ .\ f\ x$

```
zweiN :: Nat
zweiN = S (S Z)
```

$c_2 = \lambda f\ x\ .\ f\ (f\ x)$

```
dreiN :: Nat
dreiN = S (S (S Z))
```

$c_3 = \lambda f\ x\ .\ f\ (f\ (f\ x))$

- Anscheinend schreiben wir bei Church Numeralen den Term in den Rumpf des Lambda Terms und abstrahieren über die Interpretationen der Konstrukturen…

Church Numerale

- Einen Term festhalten und über die Interpretation der Konstruktoren abstrahieren? Auch das kennen wir! Das sind doch Faltungen!

```
foldNat :: b -> (b -> b) -> Nat -> b
foldNat z s Z = z
foldNat z s (S n) = s (foldNat z s n)
```

- Wenn wir die Argumente von foldNat permutieren haben wir in Haskell Church Numerale implementiert.

```
churchNat :: Nat -> (b -> b) -> b -> b
churchNat n = \f x -> foldNat x f n
```

$$c_n = \lambda f\, x\, .\, f^n(x)$$

Church Numerale

PROPOSITION (J.B. Rosser). *Define*

$$\mathbf{A}_+ \equiv \lambda xypq.xp(ypq);$$
$$\mathbf{A}_* \equiv \lambda xyz.x(yz);$$

Then one has for all $n, m \in \mathbb{N}$

(i) $\mathbf{A}_+ c_n c_m = c_{n+m}$.

(ii) $\mathbf{A}_* c_n c_m = c_{n*m}$.

Church Numerale

- Wenn Church Numerale der Faltung einer natürlichen Zahl entsprechen, wie definieren wir dann Funktionen auf Church Numeralen?
- Fangen wir mit den Konstruktoren an:

$$\text{zero} = c_0 = \lambda f\ x\ .\ x$$

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

- Wenn wir die Konstruktoren von Church Numeralen kodiert haben, dann können wir Funktionen auf diesen genauso definieren, wie wir in Haskell Funktionen mit Faltungen definiert haben.

```haskell
add :: Nat -> Nat -> Nat
--add n m = foldNat m S n
add n m = (churchNat n) S m
```

$$\text{add} = \lambda n\ m\ .\ n\ (\lambda n\ .\ \lambda f\ x\ .\ f\ (n\ f\ x))\ m$$

Church Numerale

- Aber diese Definition von der Addition von Church Numeralen sieht doch ganz anders aus, als die Definition von den Folien?
- Das ist richtig, aber wir haben in der Vorlesung bereits mehrfach darüber gesprochen, dass es keine eindeutige Repräsentation von Funktionen gibt.
- Vergleichen wir einmal beide Definitionen:

$$\text{add} = \lambda n\ m\ .\ n\ (\lambda n\ .\ \lambda f\ x\ .\ f\ (n\ f\ x))\ m$$

- Es wird nur über zwei Church Numerale abstrahiert, nicht über die Interpretationen der Konstruktoren.
- Es wird unsere Church Kodierung des Konstruktors succ zur Definition genutzt.

$$\begin{aligned} A_+ &= \lambda x\ y\ p\ q\ .\ x\ p\ (y\ p\ q) \\ &=_\alpha \lambda n\ m\ f\ x\ .\ n\ f\ (m\ f\ x) \end{aligned}$$

- Es wir explizit über die Interpretationen der Konstruktoren abstrahiert.
- Diese können nun genutzt werden. Es muss aber sichergestellt werden, dass diese auch an alle abstrahierten Church Numerale n & m richtig weitergereicht werden.

Church Numerale

add $= \lambda n\ m\,.\ n\ (\lambda n\,.\lambda f\ x\,.\ f\ (n\ f\ x))\ m$

$$\begin{aligned} A_+ &= \lambda x\ y\ p\ q\,.\ x\ p\ (y\ p\ q) \\ &=_\alpha \lambda n\ m\ f\ x\,.\ n\ f\ (m\ f\ x) \end{aligned}$$

- Beide Definitionen, sowie alle Church Kodierungen von Datenstrukturen im ungetypten λ-Kalkül sind unter der Invariante definiert, dass die Argumente selber (richtig) Church kodierte Terme der erwarteten Datenstrukturen sind.
- Diese Invariante wird in Programmiersprachen in der Regel durch Typsysteme sichergestellt.
- Das ungetypte λ-Kalkül ist aber wortwörtlich ungetypt und hat daher kein Typsystem, dass das sicherstellen kann.
- Haskell z.B. hat so ein Typsystem. Aber dazu später mehr…

Church Numerale

add $= \lambda n\ m\ .\ n\ (\lambda n\ .\ \lambda f\ x\ .\ f\ (n\ f\ x))\ m$

$$\begin{aligned} A_{+} &= \lambda x\ y\ p\ q\ .\ x\ p\ (y\ p\ q) \\ &=_{\alpha} \lambda n\ m\ f\ x\ .\ n\ f\ (m\ f\ x) \end{aligned}$$

- Der rechte Ansatz bietet sich an, um mit viel Nachdenken sehr kurze Terme zu produzieren und findet sich in den meisten Textbüchern.
- Der linke Ansatz ist genereller und lässt sich auf die meisten Datenstrukturen verallgemeinern.
- Wir werden den linken Ansatz nutzen, um uns im Folgenden anzuschauen, wie wir Haskell Datenstrukturen in Church Kodierungen übersetzen und Funktionen auf diesen definieren.

Church Kodierungen: Bool

```
data Bool = True | False
```

- Der erste Schritt sollte immer sein, die Faltung zu definieren.
- Im Fall von Bool ist die Faltung ein alter Bekannter:

```
foldBool :: a -> a -> Bool -> a
foldBool t f True = t
foldBool t f False = f
```

```
ifThenElse :: Bool -> a -> a -> a
ifThenElse b then' else' =
    foldBool then' else' b
```

- Der nächste Schritt ist, die Konstruktoren als Church Kodierung zu definieren:

true $= \lambda t\ f.\ t$

false $= \lambda t\ f.\ f$

Church Kodierungen: Bool

```
data Bool = True | False
```

true $= \lambda t\ f.\ t$

false $= \lambda t\ f.\ f$

- Dann können wir die üblichen boolschen Operatoren definieren:

```
not :: Bool -> Bool
--not b = foldBool False True b
not b = (ifThenElse b) False True
```

not $= \lambda b\,.\,b\ (\lambda t\ f.\ f)\ (\lambda t\ f.\ t)$

```
and :: Bool -> Bool -> Bool
--and l r = foldBool r False l
and l r = (ifThenElse l) r l
```

and $= \lambda l\ r\,.\ l\ r\ l$

```
or :: Bool -> Bool -> Bool
--or l r = foldBool True r l
or l r = (ifThenElse l) l r
```

or $= \lambda l\ r\,.\ l\ l\ r$

Church Kodierungen: Listen

```
data List a = Nil | Cons a (List a)
```

- Die Rechtsfaltung ist allgemeiner, als die Linksfaltung. Das heißt, dass man mit der Rechtsfaltung die Linksfaltung definieren kann, aber nicht anders herum.
- Im Fall von Listen ist mit „der Faltung“ also immer die (allgemeinere) Rechtsfaltung gemeint.
- Um die Church Kodierung von Listen zu definieren, fangen wir also mit der (Rechts-)Faltung an:

```
foldList :: b -> (a -> b -> b) -> List a -> b
foldList n c Nil = n
foldList n c (Cons a l) = c a (foldList n c l)

churchList :: List a -> b -> (a -> b -> b) -> b
churchList l = \n c -> foldList n c l
```

- Der nächste Schritt ist, die Konstruktoren als Church Kodierung zu definieren:

nil $= \lambda n\ c\,.\ n$

cons $= \lambda a\ l\ n\ c\,.\ c\ a\ (l\ n\ c)$

Church Kodierungen: Listen

```
data List a = Nil | Cons a (List a)
```

nil $= \lambda n\ c\ .\ n$

cons $= \lambda a\ l\ n\ c\ .\ c\ a\ (l\ n\ c)$

- Schauen wir uns einen Beispielterm an:

```
nullEinsZwei :: List Nat
nullEinsZwei = Cons Z (Cons (S Z) (Cons (S (S Z)) Nil))
```

nullEinsZwei $= cons\ (\lambda s\ z\ .\ z)\ (cons\ (\lambda s\ z\ .\ s\ z)\ (cons\ (\lambda s\ z\ .\ s\ (s\ z))\ nil))$

$= (\lambda a\ l\ n\ c\ .\ c\ a\ (l\ n\ c))\ (\lambda s\ z\ .\ z)\ ((\lambda a\ l\ n\ c\ .\ c\ a\ (l\ n\ c))\ (\lambda s\ z\ .\ s\ z)\ ((\lambda a\ l\ n\ c\ .\ c\ a\ (l\ n\ c))\ (\lambda s\ z\ .\ s\ (s\ z))\ (\lambda n\ c\ .\ n)))$

$\rightarrow_\beta^* \lambda n\ c\ .\ c\ (\lambda s\ z\ .\ z)\ (c\ (\lambda s\ z\ .\ s\ z)\ (c\ (\lambda s\ z\ .\ s\ (s\ z))\ n))$

Church Kodierungen: Listen

```
data List a = Nil | Cons a (List a)
```

$$\text{nil} = \lambda n\ c\,.\,n$$

$$\text{cons} = \lambda a\ l\ n\ c\,.\,c\ a\ (l\ n\ c)$$

- Funktionen auf Church kodierten Listen können wir dann als Faltungen implementieren:

```
length :: List a -> Nat
--length l = foldList Z (\a n -> S n) l
length l = (churchList l) Z (\a -> S)
```

$$\text{length} = \lambda l\,.\,l\ (\lambda s\ z\,.\,z)\ (\lambda a\,.\,(\lambda n\,.\lambda s\ z\,.\,s\ (n\ s\ z)))$$

```
map :: (a -> b) -> List a -> List b
--map f l = foldList Nil (\a x -> Cons (f a) x) l
map f l = (churchList l) Nil (\a -> Cons (f a))
```

$$\text{map} = \lambda f\ l\,.\,l\ (\lambda n\ c\,.\,n)\ (\lambda a\,.\,(\lambda a\ l\ n\ c\,.\,c\ a\ (l\ n\ c))\ (f\ a))$$

Church Kodierungen: Tipps zum Üben

Live Demo: Wie übe ich das mit Haskell?