Funktionale Programmierung
Sommersemester 2021

Prof. Dr. Jakob Rehof
TU Dortmund
LS XIV Software Engineering

Diese Vorlesung:

- DoberkatFP: Folien 148 - 195
  - Datentypen in Haskell
- Haskell-Datentypen als Church-Kodierung

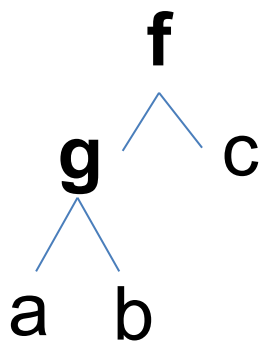
Termalgebren und Konstruktoren

- Termalgebra
  - Trägermenge: Syntaxbäume
  - Operatoren: Syntaxkonstruktoren
- Intuitiv kann man *Konstruktoren* als abstrakte Funktionssymbole auffassen, die keine weiteren Berechnungsregeln haben als die Formierung von syntaktischen Ausdrücken

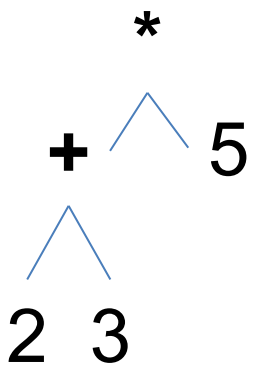
Syntax und Semantik

- Links: Term mit zweistelligen Funktionssymbolen f und g und nullstelligen Funktionssymbolen a, b und c
- Rechts: Interpretierter Term mit Interpretation {f -> (*); g -> (+); a -> 2; b -> 3; c -> 5}

„**f**(**g**(a,b), c)“

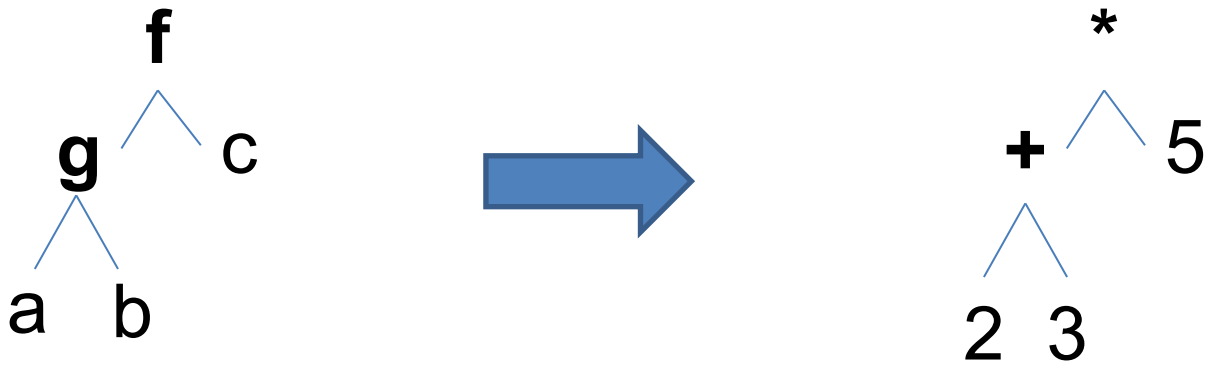


„(2 **+** 3) * 5“



Termalgebren und Konstruktoren

- Eine Möglichkeit mit Termen zu „rechnen“, also Funktionen über diesen zu definieren, ist also eine Interpretation für die Konstruktoren anzugeben.
- Die Interpretation selbst ersetzt dann einfach jeden Konstruktor durch die entsprechende Funktion.



Termalgebren und Konstruktoren

- Ein Beispiel für Terme über Datentypen, die wir bereits kennen:
- Der Term 1 : 2 : 3 : [] und die Interpretation {(:) -> (+); [] -> 0} der Konstruktoren liefert uns:

$$1 + 2 + 3 + 0 = 6$$

- Kommt Ihnen das bekannt vor? Wie würde man eine Funktion definieren, die einen Term interpretiert?

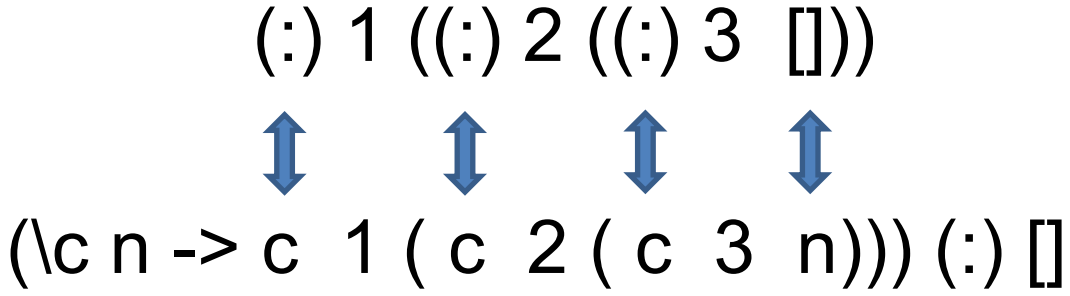
Termalgebren und Church-Kodierung

- Die Idee der Church-Kodierung von Datentypen ist nun relativ einfach:

Schreibe die Struktur des Terms und abstrahiere über die Konstruktoren.

Termalgebren und Church-Kodierung

- Wir erinnern uns:
- (\c n -> c 1 (c 2 (c 3 n))) entspricht der Liste 1:2:3:[], wenn wir (:) und [] als Argumente übergeben.



Haskell-Datentypen und Church-Kodierung

- Auf diese Weise lassen sich Haskell-Datentypen und Church-kodierte Datentypen im Lambda-Kalkül beliebig hin und her übersetzen.
- Nutzen wir das doch, um herauszufinden, wie man Nat, den Typ der natürlichen Zahlen in Haskell definieren kann.

Haskell-Datentypen und Church-Kodierung

- Church-Numerale werden wie folgt verwendet:
  - 0 = \s z -> z
  - 1 = \s z -> s z
  - n = \s z -> s^n z
- Wir brauchen also zwei Konstruktoren, wovon einer die Stelligkeit 0 und einer die Stelligkeit 1 hat.

Haskell-Datentypen und Church-Kodierung

- Mit diesem Wissen können wir schon den Haskell-Datentyp schreiben:

```haskell
data Nat = Zero | Succ Nat deriving Show

test0 :: Nat
test0 = Zero

test1 :: Nat
test1 = Succ Zero

add :: Nat -> Nat -> Nat
add Zero n = n
add (Succ n) m = Succ (add n m)
```

- Neben der Stelligkeit der Konstruktoren sind für Haskell-Datentypen natürlich auch die Argumenttypen wichtig und teilweise etwas schwerer aus der Church-Kodierung zu übersetzen.

Haskell-Datentypen und Church-Kodierung

- Church-kodierte Datentypen sind also Terme, bei denen sozusagen über die Interpretation der Konstruktoren Lambda-abstrahiert wird.
- Die Church-Kodierung entspricht also einfach dem Lambda-Term der (Rechts-)Faltung eines Haskell-Datentyps.

Haskell-Datentypen und Church-Kodierung

```haskell
natToChurch :: Nat -> (a -> a) -> a -> a
natToChurch Zero = \s z -> z
natToChurch (Succ n) = \s z -> s (natToChurch n s z)


listToChurch :: [a] -> (a -> b -> b) -> b -> b
listToChurch [] = \ c n -> n
listToChurch (x : xs) = \c n -> c x (listToChurch xs c n)
```

- Kommt Ihnen listToChurch bekannt vor?