universität innsbruck

WS 2025/2026

Funktionale Programming

Woche 13 – Lambda-Kalkül, Zusammenfassung, Fragestunde

René Thiemann
Philipp Dablander Joshua Ocker Michael Schaper Lilly Schönherr Adam Pescoller

Institut für Informatik

Letzte Vorlesung

- zyklische Definitionen, z.B. `fibs = 0 : 1 : zipWith (+) fibs (tail fibs)`
- abstrakte Datentypen
  - Spezifikation von Funktionen und Verhalten
  - Implementierungs-Details bleiben verborgen (mittels geeigneter Export-Liste des Moduls)
  - Beispiel: Warteschlangen
    - benötigt, um etwa Breitensuche in Bäumen zu implementieren
    - erste Implementierung war einfach, n Operationen benötigen bis zu $\sim \frac{1}{2}n^2$ Schritte
    - verbesserte Implementierung repräsentiert Warteschlangen als Paar von zwei Listen, n Operationen benötigen maximal $\sim 2n$ Schritte

λ-Kalkül

Einführung zum λ-Kalkül

- λ-Kalkül arbeitet mit λ-Termen, die entweder eine λ-Abstraktion, eine Variable oder eine Funktionsanwendung sind
- minimalistische Sprache, es gibt keine Typen, keine Datentypen, keine Funktions-Definitionen, keine eingebaute Arithmetik, ...
- einziger Auswertungs Mechanismus: β-Reduktion

  ersetze `(\ x -> s) t` durch `s[x/t]`

  wobei `s[x/t]` der Term ist, der entsteht, wenn man in `s` die Variable `x` durch `t` substituiert
- hinreichend Ausdrucks-stark, um funktionale Programme zu simulieren

Boolesche Werte im λ-Kalkül

- Kodierung der Booleans als λ-Terme, d.h., wir implementieren `Bool` als abstrakten Datentyp
  - interne Konstruktion wird wie folgt durchgeführt
    - `Bool`: `a -> a -> a`
    - `True`: `\ x y -> x`
    - `False`: `\ x y -> y`
    - `if-then-else`: `\ c t e -> c t e`
  - erfüllte Axiome
    - `(if True then t else e) = t`:

```
  (\ c t e -> c t e) (\ x y -> x) t e
= (\ t e -> (\ x y -> x) t e) t e
= (\ e -> (\ x y -> x) t e) e
= (\ x y -> x) t e
= (\ y -> t) e
= t
```

    - `(if False then t else e) = e`: analog

Boolesche Werte im λ-Kalkül, fortgesetzt

- bislang haben wir λ-Terme zur Repräsentation von `True`, `False`, und `if-then-else`
- andere Boolesche Funktionen sind nun leicht erhältlich
  - `b && c = if b then c else False`
  - `b || c = if b then True else c`
  - `not b  = if b then False else True`
- Beispiel: Berechnung von `False && True`:

```
  False && True                  -- unfold encoding of &&
= if False then True else False  -- unfold encoding of ite, False, True
= (\ c t e -> c t e) (\ x y -> y) (\ x y -> x) (\ x y -> y)
  -- the line above is the lambda-term that is evaluated
= (\ t e -> (\ x y -> y) t e) (\ x y -> x) (\ x y -> y)
= (\ e -> (\ x y -> y) (\ x y -> x) e) (\ x y -> y)
= (\ x y -> y) (\ x y -> x) (\ x y -> y)
= (\ y -> y) (\ x y -> y)
= \ x y -> y                     -- representation of False
```


Paare im λ-Kalkül

- Paare können ähnlich zu Booleschen Werten kodiert werden
- wir benötigen drei Operationen: `(x, y)`, `fst`, `snd`
  - Kodierung von Paaren `(x :: a, y :: b)` mit $a \neq b$ ist nicht Typ-korrekt in Haskell!
  - Kodierung von `(x, y)`: `\ c -> if c then x else y`
  - Kodierung von `fst`: `\ p -> p True`
  - Kodierung von `snd`: `\ p -> p False`
- Korrektheit, z.B. `snd (x, y) = y`

```
snd (x, y)                                  -- expand snd and (x, y)
= (\ p -> p False) (\ c -> if c then x else y)          -- beta
= (\ c -> if c then x else y) False                     -- beta
= if False then x else y                    -- soundness of ite
= y
```

- mit Paaren kann man auch Tupel und Listen modellieren

Church-Numerale

- auch natürliche Zahlen können im λ-Kalkül repräsentiert werden
- Church-Numerale: n wird als `\ f x -> f (f ... (f x) ...)` kodiert, wobei es n Anwendungen von `f` gibt
- Kodierung des Typs der natürlichen Zahlen: `(a -> a) -> a -> a`
- Beispiele
  - `0`: `\ f x -> x`
  - `1`: `\ f x -> f x`
  - `2`: `\ f x -> f (f x)`
  - `(== 0)`: `\ n -> n (\ b -> False) True`
  - `(+ 1)`: `\ n f x -> f (n f x)`
  - `(+)`: `\ n m f x -> n f (m f x)`
  - `(*)`: `\ n m f x -> n (m f) x`
  - `\ x -> x - 1`: möglich, aber schwieriger

Rekursion

- um allgemeine Rekursion zu definieren, kann man den `Y`-Kombinator verwenden:
  `Y = \ f -> (\ x -> f (x x)) (\ x -> f (x x))`
- wichtige Eigenschaft: `Y g` wertet zu `g (Y g)` aus,
  d.h., `Y g` ist Fixpunkt von `g`: `g (Y g) = Y g`
- rekursive Funktionen können als Fixpunkte von nicht-rekursiven Funktionen definiert werden

```
add x y = if x == 0 then y else add (x-1) (y+1)
-- add is fixpoint of the non-recursive function addNR
-- equality: addNR add = add
addNR a x y = if x == 0 then y else a (x-1) (y+1)
```

- Kodierung der obigen Additions-Funktion im λ-Kalkül
  - kodiere nicht-rekursive Funktion `addNR` als λ-Term `t` wie auf vorigen Folien
  - kodiere `add` als Fixpoint: `add` = Fixpunkt von `addNR` = `Y t`

Zusammenfassung des Kurses

Was Sie gelernt haben sollten

- Definition von Typen und Funktionen
  - Typ-Definitionen mittels `type`, `newtype`, und `data`
  - Definition von Funktionen mit verschiedensten Techniken: Pattern Matching, Rekursion, Kombination von vordefinierten Funktionen (höherer Ordnung), List Comprehensions, ...
- Verständnis von Typen
  - parametrischer Polymorphismus, Typ-Variablen und Typ-Klassen
  - Fähigkeit, den allgemeinsten Typen für einfache Definitionen zu bestimmen
- I/O in Haskell, do-Notation, Kompilierung mit `ghc`
- Definition und Vorteile von Modulen und abstrakten Datentypen
- Auswertungs-Strategien, insbesondere Haskells Lazy Evaluation
- Kenntnis mehrerer Typen und Funktionen der `Prelude`
  - Typen `Int`, `Integer`, `Double`, `[a]`, `Maybe a`, `Either a b`, `String`, `Char`, `Bool`, Tupel
  - Typklassen für Zahlen, `Show`, `Read`, `Eq`, `Ord`
  - Arithmetische und Boolesche Funktionen und Operatoren
  - Funktionen für Listen und Strings
  - I/O: Funktionen um Daten einzulesen und auszugeben (auch mittels Dateien)

Was Sie nicht in diesem Kurs gelernt haben

- Algorithmus zur Bestimmung des allgemeinsten Typs (Typinferenz)
- Compilierung von funktionalen Programmen
- statische Analyse und Optimierung von funktionalen Programmen
- Debugging und Verifikation von funktionalen Programmen
- Erstellung von nebenläufigen und parallelen funktionalen Programmen
- weitere funktionale Programmierungs-Techniken (Monaden, Funktoren, Continuations, Lenses, ...)

Weiterführende Kurse

- Fortgeschrittene Funktionale Programmierung (Bachelor oder Master)
- Programmverifikation (Bachelor)
- Interaktives Theorem Beweisen mit Isabelle/HOL (Master)