Tutorium 05: λ-Kalkül

Paul Brinkmeier

18. November 2019

Tutorium Programmierparadigmen am KIT

Heutiges Programm

Programm

- Übungsblatt 4
- λ-Kalkül: Basics + Church-Zahlen

Programm

- Übungsblatt 4
- λ-Kalkül: Basics + Church-Zahlen
- λ-Kalkül in Haskell

Übungsblatt 4

2.1, 2.3 — AST: Datenstruktur

```
module AstType where


data Exp t
  = Var t
  | Const Integer
  | Add (Exp t) (Exp t)
  | Less (Exp t) (Exp t)
  | And (Exp t) (Exp t)
  | Not (Exp t)
  | If (Exp t) (Exp t) (Exp t)
```

- `t` ist Typvariable, um bspw. `Int`s als Namen zuzulassen
- Das kommt bspw. bei Compiler-Optimierungen zum Einsatz

2.2 — AST: Auswertung

```
module AstEval where
import AstType

type Env a = a -> Integer

eval :: Env a -> Exp a -> Integer
eval env (Var v) = env v
eval env (Const c) = c
eval env (Add e1 e2) = eval env e1 + eval env e2
```

2.3 — AST: Boolsche Ausdrücke

```
module AstEval2 where

eval :: Env a -> Exp a -> Integer
eval env (Less e1 e2) = b2i $
  (eval env e1) < (eval env e2)
eval env (And e1 e2) = b2i $
  (i2b $ eval env e1) && (i2b $ eval env e2)
eval env (Not e) = b2i $ not $ i2b $ eval env e

b2i b = if b then 0 else 1
i2b i = if i == 0 then False else True
```

- Aufgabe sorgfältig lesen, nur 0 ist „falsey“ in C
- $\rightsquigarrow$ kann einem in der Klausur in den Arsch beißen

2.4 — AST: Show

```
module AstShow where
import AstType

instance Show t => Show (Exp t) where
  show (Const c) = show c
  show (Var   v) = show v -- Darf man wegen Show t
  show (Add a b) =
    "(" ++ show a ++ " + " ++ show b ++ ")"
-- etc.
```

- `Show t => Show (Exp t)` $\Leftrightarrow$ „Wenn man `ts` anzeigen kann, kann man auch `Exp ts` anzeigen“

Wiederholung

Algebraische Datentypen

```
module DataExamples where

data Bool = True | False

data Category = Jackets | Pants | Shoes
  deriving Show
data Filter
  = InSale
  | IsCategory Category
  | PriceRange Float Float
```

- Keyword data definiert *neuen* Typ
- „enum auf Meth“

Typklassen

```
module TypeClassExamples where

-- Ersatz für null in C-likes
-- Auch bekannt als "Maybe"
data Optional a = Present a | NoValue

instance Show a => Show (Optional a) where
  show (Present x) = show x
  show NoValue     = "null"
```

- Typklassen stellen globale Operationen für Typen bereit
- Bspw. `Eq` und `Ord` für Vergleiche, `Enum` für Aufzählbarkeit

λ-Kalkül

λ-Terme

Ein Term im λ-Kalkül hat eine der drei folgenden Formen:

Notation	**Besteht aus**	**Bezeichnung**
x	x : Variablenname	Variable
$\lambda p.b$	p : Variablenname b : λ-Term	Abstraktion
$f\ a$	f, a : λ-Terme	Funktionsanwendung

- „λ-Term“: rekursive Datenstruktur
- Semantik definieren wir später

λ-Terme

Ein Term im λ-Kalkül hat eine der drei folgenden Formen:

Notation	**Besteht aus**	**Bezeichnung**
x	x : Variablenname	Variable
$\lambda p.b$	p : Variablenname b : λ-Term	Abstraktion
$f\ a$	f, a : λ-Terme	Funktionsanwendung

- „λ-Term“: rekursive Datenstruktur
- Semantik definieren wir später
- Jetzt: Ergänzt das Modul `Lambda` um die fehlenden Typen
  - +Fragen zur ÜB-Korrektur

λ-Terme in Haskell

```
module Lambda where

data LambdaTerm
  = Var String  -- Variable
  | App () ()   -- Funktionsanwendung: f a
  | Abs () ()   -- Abstraktion: \p.b
```

- //github.com/pbrinkmeier/pp-tut
- Modul `x` liegt in `slides/demos/x.hs`