Funktionale Programmierung

Grundlagen

Julian Müller

February 7, 2019

Warum funktionale Programmierung?

Funktionaler “Hype”

- C++: Templates, Template Metaprogramming, Lambdas
- Java: Generics, Lambdas, Streams, MapReduce
- C Sharp: Generics, Lambdas, LINQ
- Scala: Funktionale Programmiersprache auf der JVM (Apache Spark, . . .)
- JavaScript: “Web-Assembler”, Scheme in Java’s clothing
- R: Data Science und Machine Learning
- Abap: . . . Abap?

Ernsthaft Abap?

- Type-Inference (#)
- Konstruktorausdrücke
- Streaming

Abap Konstruktorasudrücke

```
... REDUCE type(
      [let_exp]
      INIT {x1 = rhs1}|{<x1> = wrexpr1}
                      |{x1|<x1> TYPE dtype1}
           {x2 = rhs2}|{<x2> = wrexpr2}
                      |{x2|<x2> TYPE dtype2}
           ...
      FOR for_exp1
      FOR for_exp2
      ...
      NEXT ...
           {x1 = rhs1}|{<x1> = wrexpr1}
           {x2 = rhs2}|{<x2> = wrexpr2}
           ... ) ...
```

Abap Konstruktorausdrücke

```
cl_demo_output=>display(
  REDUCE i( INIT s = 0
            FOR  i = 1 UNTIL i > 10
            NEXT s = s + i ) ).
```

Funktionaler “Hype”: Buzzwords

- Lambdas (anonymous functions)
- Closures
- Currying / Higher Order Functions
- Immutability
- Streaming
- Map-Reduce
- Monaden
- ...

Anwendungen funktionaler Programmierung

Massiv parallelisierte Anwendungen

- Backend Services: Sigma (facebook Spamfilter)
- Cluster Computing: Apache Spark
- High Frequency Trading: Jane Street, Allston Trading
- Telekommunikation / Switches (hier isb. Erlang)
- ...

Language and DSL Design (I)

- Purescript: Functional Web Programming
- Idris: Funktionale Sprache mit Dependent Types
- Crush: Untypisiertes Lambdakalkül
- Pandoc: Markup Converter (diese Präsentation)
- ...

Tests und Beweis von Programmeigenschaften

- Coq: Beweisassistent mit Programmsynthese
- Idris: Funktionale Sprache mit Dependent Types
- Haskell: Beweis via SMT (Liquid Haskell), Property-Based-Testing
- ...

Livedemo: DSL der Arithmetik

Livedemo

Sprache der Arithmetik

- 100 Zeilen Code
- Sprachdefinition: Variablen, Addition, Multiplikation
- Parser
- Fold (-> Constructor Expression)
- Pretty Printer
- Evaluator
- Term Rewriting / Simplification: Distributivität
- Property-Based-Testing

Livedemo

Kleiner Cheat

- Bibliothek für den Parser (Parsec)
- Bibliothek für das Property-Based-Testing (QuickCheck)

Ohne jede Bibliothek wären es 200 Zeilen ...

Livedemo

Was kann man mit so einer DSL machen?

- Verwendung in Businessrules, Auswertung zur Compilezeit
- Symbolische Algebra
  - Automatische Differenziation
  - Automatische Integration (Risch-Algorithmus)
- Verwendung in Beweissystemen (Presburgerarithmetik)
- ...

Was ist funktionale Programmierung?

Funktionale Programmierung im engeren Sinn

Funktionale Programmierung im engeren Sinn:

> *Alle Programmiermodelle denen das Lambdakalkül zugrunde liegt*

Funktionale Programmierung im weiteren Sinn

Funktionale Programmierung im weiteren Sinn:

> *Alle Programmiermodelle die an das Lambdakalkül angelehnt sind*

Das untypisiert / pure Lambdakalkül

Komponenten des Lambdakalküls

Jede Programmiersprache besteht aus zwei Komponenten:

1. Syntax zur Konstruktion von Programmen
2. Semantik zur Auswertung der Programme

Das gilt auch für das Lambdakalkül

Definition Lambdakalkül - Generative Grammatik

Die Menge aller Lambdaterme *Term* wird durch die folgende generative Grammatik definiert:

$$Term \;=\; Var \mid (Term\ Term) \mid (\lambda Var.Term)$$

wobei die folgenden Symbole verwendet werden:

1. (Variablen) $Var = v_0, v_1, \ldots$
2. (Abstraktor) λ
3. (Klammern) (,)

Hintergrund

Das ist die gesamte Syntax der Programmiersprache Lambdakalkül.

Wie kommt man darauf so etwas zu entwickeln?

Geschichte des Lambdakalküls

Geschichte: Geschichte der Mathematik

Die Geschichte der Mathematik ist eine Geschichte grandiosen Scheiterns

- Cantor: "naive" Mengenlehre (Russellsche Antinomie)
- Frege: Prädikatenlogik zweiter Stufe (Russellsche Antinomie)
- Hilbert: Widerspruchsfreiheit der Arithmetik (Gödels Unvollständigkeitssatz)
- Hilbert: Entscheidungsproblem (?)

Geschichte: Entscheidungsproblem

Kernfrage der Logiker und Mathematiker -
Entscheidungsproblem (Hilbert 1928)

> *Gibt es einen Algorithmus [. . .], der von einer beliebigen Formel eines logischen Kalküls feststellt, ob sie aus gewissen vorgegebenen Axiomen folgt oder nicht?*

Der Algorithmus muss also effektiv berechenbar sein

Geschichte: Entscheidungsproblem

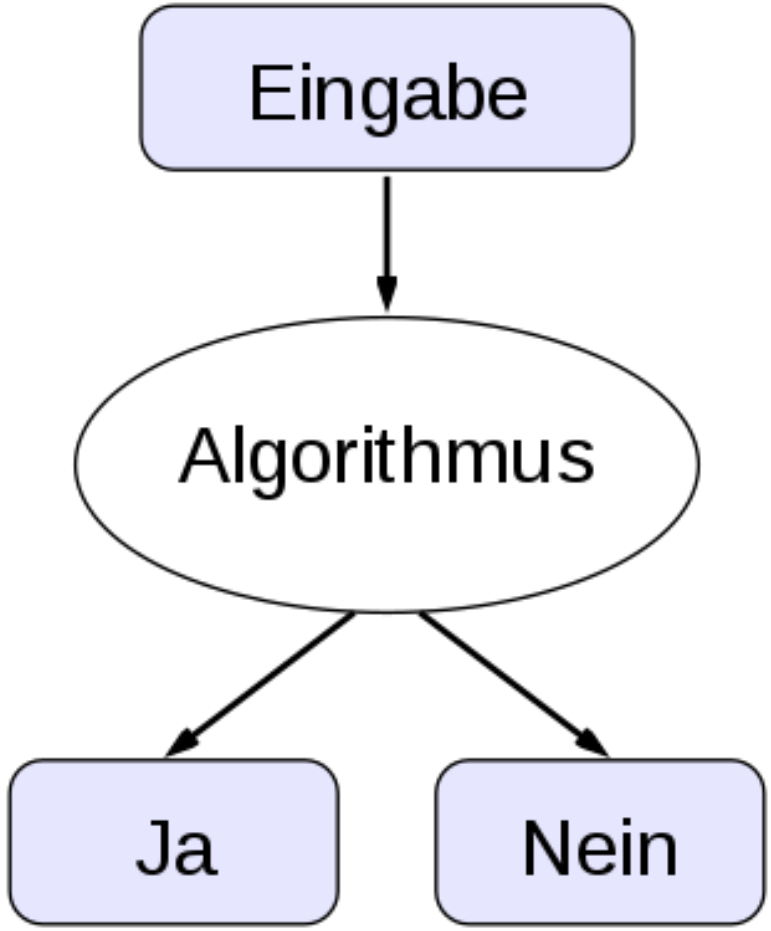

Figure: Entscheidungsproblem

Geschichte: Entscheidungsproblem

Geschichte: Algorithmen

Aber was ist ein Alogrithmus? Und was bedeutet Berechenbarkeit?

Intuitiv: Algorithmen sind berechenbare Funktionen

Drei Formalisierungen berechenbarer Funktionen:

1. Rekursive Funktionen (Kurt Gödel)
2. Turingmaschinen (Alan Turing)
3. Lambdakalkül (Alonzo Church)

Geschichte 3: Church-Turing-These

Church-Turing-These

> *Die Klasse der Turing-berechenbaren Funktionen stimmt mit der Klasse der intuitiv berechenbaren Funktionen überein*

Weiter gilt im Bezug auf die Klasse der formalisierbaren Funktionen:

Turing-Maschinen = Lambda-Kalkül = Rekursive Funktionen

Ein und derselbe Gegenstand (Definition der berechenbaren Funktion) aus drei verschiedenen Perspektiven.

Geschichte: Abstraktion / Essenz

Geschichte: Die Essenz berechenbarer Funktionen

Syntax des Lambdakalküls

$$
\begin{aligned}
\mathit{Term} &= \mathit{Var} && (1)\\
&\mid (\mathit{Term}\ \mathit{Term}) && (2)\\
&\mid (\lambda \mathit{Var}.\mathit{Term}) && (3)
\end{aligned}
$$

wobei

1. Variablen
2. Funktionen anwenden (lies: $(\mathit{Fun}\ \mathit{Arg})$)
3. Funktionen definieren

Geschichte: Entscheidungsproblem

Das grandiose Scheitern setzt sich fort ...

- 1930: Hilbert glaubt nicht an die Existenz unlösbarer Probleme
- 1936: Alonzo Church beweist, dass die semantische Äquivalenz zweier Lambdaterme (= Programme) im Allgemeinen nicht beweisbar ist
- 1936: Alan Turing beweist, dass das Halteproblem formuliert mit Turingmaschinen nicht lösbar ist

Syntax des Lambdakalküls

Syntax des Kalküls: Interpretierte Konstanten

Out-of-the-Box gibt es keine weiteren Datentypen wie `String`, `Bool` oder `Int`

- Diese lassen sich im Lambdakalkül kodieren (vgl. Church-Encoding)
- Für die bessere Verständlichkeit nehmen wir ab hier an einige dieser Konstanten bereits definiert zu haben (+, 1, 2, 3, ...)

Syntax des Kalküls: Abstraktion

Funktionen werden durch den Abstraktionsoperator λ gebildet. Die Anatomie einer Funktion ist dabei wie folgt:

$$\lambda input.body$$

Beispiel

$$(\lambda x.x^2 + x + 1)$$

wobei

- Input (Signatur) der Funktion: x
- Funktionskörper: $x^2 + x + 1$

Syntax des Kalküls: Abstraktion

$$f(x) = x^2 + x + 1$$

wird wie folgt als Lambdaterm dargestellt:

$$\lambda x.x^2 + x + 1$$

- Die frei im Term $T(x) = x^2 + x + 1$ vorkommende Variable x wird durch die Abstraktion $\lambda x.T(x)$ gebunden
- Die Variable x in T ist jetzt im Skopus der Abstraktion λx
- Ein Lambdaterm ohne freie Variablen wird *Kombinator* genannt

Syntax des Kalküls: Applikation

Die Anwendungen der Funktion wird durch die *Applikation* formalisiert:

$$(\lambda x.x^2 + x + 1)\ 3$$

Durch Applikation wird nur ein neuer Term gebildet – die Berechnung findet noch nicht statt

Einige Lambdaterme im puren Lambdakalkül

x

$x(\lambda x.\lambda y.z)h$

$\lambda z.(\lambda x.y(\lambda x.z))(\lambda y.y)$

$S = \lambda x.\lambda y.\lambda y.x\ z\ (y\ z)$

$K = \lambda x.\lambda y.x$

$I = \lambda x.x$

$\omega = \lambda x.x\ x$

$\Omega = \omega\ \omega = (\lambda x.x\ x)(\lambda x.x\ x)$

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

Semantik des Lambdakalküls

Semantik des Kalküls: Auswertung / Reduktion

- Lambda-Terme werden durch *Substitution* ausgewertet
- Variablen im Skopus einer Abstraktionen bezeichnen die Stellen, an denen bei der Substitution die Funktionsargumente eingesetzt werden:

$$\cfrac{\cfrac{\cfrac{\cfrac{(\lambda x.x^2 + x + 1)\ 3}{3^2 + 3 + 1}\,\text{(Substitution)}}{9 + 3 + 1}\,\text{(Arithmetik)}}{12 + 1}\,\text{(Arithmetik)}}{13}\,\text{(Arithmetik)}$$

Curch Encoding: Booleans

```
let tru = \p.\q. p in
let fls = \p.\q. q in
let ite = \p.\x.\y. p x y in
let neg = \p. ite p fls tru in
neg tru
```

Ausführen:

```
cat .\examples\lambda\church.lambda | crush
```

Auswertung von Lambdatermen: Omega

Das Lambdakalkül ist Turingvollständig - nicht jede Auswertung eines Terms terminiert.

```
let omega = \x. x x in
let Omega = omega omega in
Omega
```

Ausführen:

```
cat .\examples\lambda\omega.lambda | crush --limit 5
```

Currying und Higher-Order-Functions

Beispiel Addition

Mehrstellige Funktionen können durch einstellige Funktionen dargestellt werden:

Die Additionsfunktion

$$f(x, y) = x + y$$

dargestellt im Lambda-Kalkül:

$$\lambda x.\lambda y.x + y$$

Partielle Applikation der Addition I

Die "zweistellige" Funktion $\lambda x.\lambda y.x + y$ wird durch Applikation auf die Zahl 1 zur "einstelligen" Funktion $\lambda y.1 + y$:

$$\frac{(\lambda x.\lambda y.x + y)\ 1}{(\lambda y.1 + y)}\,(\text{Substitution})$$

Partielle Applikation der Addition I

Die Funktion $\lambda y.1 + y$ kann dann auf ein weiteres Argument angewendet werden um einen Wert zu errechnen:

$$\frac{\frac{(\lambda y.1 + y)\ 4}{1 + 4}\,(\text{Substitution})}{5}\,(\text{Arithmetik})$$

Partielle Applikation der Addition I(als Graph)

x	y	x+y
0	6	6
0	7	7
0	8	8
0	9	9
1	0	1
1	1	2
1	2	3
2	0	2
2	1	3
2	2	4

Figure: Addition

Partielle Applikation der Addition II(als Graph)

A	B	C
x	y	x+y
1	0	1
1	1	2
1	2	3
1	3	4
1	4	5
1	5	6
1	6	7
1	7	8
1	8	9
1	9	10

Figure: Addition

Currying und Factories

Die Funktion $\lambda x.\lambda y.x + y$ kann auch als kleine Factory für Inkrementfunktionen betrachtet werden.

Definition

```
-- this is the function definition in haskell
incrementorFactory = \x ->\y -> x + y

addOne = incrementFactory 1
addTwo = incrementFactory 2
addThree = incrementFactory 3
```

und Auswertung:

```
addTow 7
>> 9
```

Haskell Syntax

Anmerkung: Das erste Mal Haskell! In Haskell wird

- \ statt λ und
- der Pfeil -> statt .

für die Abstraktion verwendet

First Class Functions

Funktionen sind "First-Class-Citizens" in funktionalen Programmiersprachen. Sie können an Funktionen übergeben werden oder von Funktionen zurückgegeben werden, in Datenstrukturen gespeichert und On-The-Fly generiert werden. Es handelt sich ja nur um Lambdaterme ...

- Funktion = Lambdaterm
- Funktionales Programm = Lambdaterm
- Ausführung eines Funktionalen Programms: Evaluation durch Substitution

Motivation Typensystem

Das pure Lambdakalkül kennt keine Typen; es wird daher auch untypisiertes Lambdakalkül genannt.

Das hier ist problemlos möglich:

```
addInt = (\x. \y. x +_int y)

program = addInt True "hello world"
```

Was passiert?

Die Eingabewerte sind nicht in der Definitionsmenge der Funktion enthalten (es sind ja keine Ganzzahlen).

Es kommt zu einem Laufzeitfehler bzw. Stuck-Terms

```
True +_int "hello world"
```

oder noch schlimmer: Das Programm läuft weiter und behandelt den Input wie Zahlen (soweit es geht)

Das (einfach) typisierte Lambdakalkül

Die Spezifikation der Funktion wird durch Typen ausgedrückt

```
addInt = (\x:Int.\y:Int. x+y): Int -> Int -> Int
```

wobei `Value : Type` die Typisierungsrelation darstellt.

Funtkionen und Typen in Haskell

Funktionsdefinition in Haskell

```
addInt :: Int -> Int -> Int
addInt x y = x + y
```

Funtkionen und Typen in Haskell

Anwendung

```
addInt True "Hello"
```

Fehler zur Compilezeit:

```
<interactive>:3:8: error:
    * Couldn't match expected type
        'Int' with actual type 'Bool'
    * In the first argument of 'addInt',
        namely 'True'
      In the expression: addInt True "Hello"
      In an equation for 'it':
        it = addInt True "Hello"
```

Ende des ersten Teils

Ausblick

- Typen: Summentypen, Produkttypen
- Parametrischer Polymorphismus (Generics)
- Ad-Hoc Polymorphimus (Overloading)
- Curry-Howard-Isomorphism

Links

- Haskell
- Stack (Haskell Build Tool)
- Learn Yourself a Haskell for Great Good!
- Real World Haskell
- State of the Haskell ecosystem
- What I Wish I knew when learning Haskell
- Haskell Reddit

Install Crush

Zum installieren von `Crush`

- github: https://github.com/julmue/crush (hier auch README)
- `git clone https://github.com/julmue/crush`
- Öffne Powershell / Git Bash im Verzeichnis
- `stack install`
- evtl. Installationsverzeichnis der Pfadvariable hinzufügen
- Beispiel: cat ./examples/cooked.lam | crush