Funktionale Programmierung

Lambda Kalküle

Woche	Thema	Praktika
15	Syntactic Sugar I: Active Patterns	
16	Syntactic Sugar II: Sequenzen und Lazy Listen	
17	Funktionale Muster I: Railway oriented programming	
18	Funktionale Muster II: Monade	
19	Theoretische Fundierung I: λ-Kalküle	
20	Theoretische Fundierung II: Typsysteme, Hindley-Milner	
21	RESERVE	

- Der Lambda-Kalkül als funktionale “Proto-Sprache”
  - Terme
  - Variablen und Substitution
  - Reduktionen und Normalformen
- Implementation mit F#
  - Parsen (wenn die Zeit reicht)
  - Auswerten (interpretieren)

Erinnerung:

> *Functional programming is [. . .] evaluation of expressions, rather than execution of commands [. . .]*

- Sie kennen mathematische Modelle, die den prozeduralen Ansatz formalisieren (Registermaschinen, Turingmaschinen, Kellerautomaten, Protosprachen: WHILE, GOTO)
- Wie sieht eine funktionale “Proto-Sprache” aus?

λ Kalküle:

- Implementieren die Idee von Algorithmen als "evaluation of expressions"
- Einfache aber ausdrucksstarke Syntax
- Mächtige Semantik (Turing vollständig)
- Modelliert Funktionsdefinitionen (Abstraction) und Funktionsanwendung (Evaluation)
- Implementiert die "programs as data duality" (höhere Funktionen)
- Funktionale Sprachen können als Variante gesehen werden

Lambda Kalküle bestehen aus folgenden “Zutaten”:

- Terme (“Programme” im λ-Kalkül)
- Regeln zur Manipulation von Termen
- Eventuell ein Typensystem (typisierte Kalküle)

Der Zeichenvorrat (Alphabet) mit dem wir λ-Terme bauen wollen besteht aus unendliche vielen Variablen $x, y, z, v, v_1, x_1, ..$, dem Zeichen λ, dem Punkt und Klammern. Fakultativ können dem Alphabet beliebig Konstanten hinzugefügt werden.

λ-Terme sind folgendermassen gegeben:

- Jede Variable und jede Konstante ist ein Term.
- Sind A und B Terme, dann ist auch $(A\ B)$ ein Term (Applikation).
- Ist x eine Variable und A ein Term, dann ist auch $\lambda\, x.A$ ein Term (Abstraktion).

Bemerkungen:

- Ein Term von der Form $(A\ B)$ steht für die Anwendung von A auf B. Wie z.B. in F# der Ausdruck `(f x)` für das Resultat der Anwendung von `f` auf `x` steht.
- Ein Term $\lambda\, x.A$ entspricht einer anonymen Funktion `fun x -> A`. Aufgrund der Typisierung von F# lassen sich nicht alle Terme (des untypisierten λ-Kalküls) in F# realisieren (Bsp.: $\lambda\, x.(x\ x)$).

Implementieren Sie einen Typ `Term` entsprechend der Vorlage unten in F#, der λ-Terme repräsentiert. Es sollen dabei folgende Vorgaben implementiert werden:

- Implementieren sie Variablen als beliebige Strings.
- Folgende Konstanten sollen implementiert werden:
  - Zahlen (`int`)
  - Funktionsterme: `add`, `IsZero`

```
type Term =
    | .......// Constant functions
    | N .....// Constants for numbers
    | V .....// Variables
    | App ...// Application
    | Lbd ...// Abstraction
```

```
type Term =
    | Add | IsZero
    | N of int
    | V of string
    | App of Term * Term
    | Lbd of string * Term
```

Aufgabe

Implementieren Sie eine Funktion `print`, die Objekte vom Typ `Term` als String repräsentiert.

Beispielausgabe für

```
Lbd ("x",Lbd ("y",App (App (Add,V "x"),V "y")))
```

ist:

```
string = "λx.λy.((Add x) y)"
```

```
let rec print = function
    | N x
        -> sprintf "%i" x
    | V x
        -> x
    | App (a,b)
        -> sprintf "(%s %s)" (print a) (print b)
    | Lbd (x,b)
        -> sprintf "L%s.%s" x (print b)
    | x -> sprintf "%A" x
```

Einige Konventionen bezüglich der Schreibweise von Termen:

- Äussere Klammern können weggelassen werden
  - A wird als (A) gelesen.
- Applikation ist linksassoziativ.
  - $A\ B\ C$ wird als $((A\ B)\ C)$ gelesen.
- Der Bereich einer "quantifizierten Variable" wird grösstmöglich angenommen.
  - $\lambda x.A\ B\ C$ wird als $\lambda x.((A\ B)\ C)$ gelesen.
- Terme von der Form $\lambda xyz.A$ werden als $\lambda x.\lambda y.\lambda z.A$ gelesen.

Aufgabe

Schreiben Sie den Term $\lambda xy.AB\,\lambda u.A$ aus.

Die Menge der freien Variablen $FV(A)$ eines λ-Terms A ist wie folgt gegeben:

- Ist A eine Konstante, dann ist $FV(A) = \emptyset$.
- Ist A eine Variable v, dann gilt $FV(A) = \{v\}$.
- Ist $A = (B\ C)$, dann ist $FV(A) = FV(B) \cup FV(C)$.
- Ist $A = \lambda x.B$, dann ist $FV(A) = FV(B) \setminus \{x\}$.

Variablen, die in einem Term vorkommen aber nicht frei sind heissen gebundene Variablen.

Aufgabe

Implementieren Sie die Funktion FV in F#.

- Die Funktion soll den Typ `Term -> Set<string>` haben.
- Testen Sie Ihre Funktion am Term $(y\ (\lambda\, x.Add))$.

```
let freeVar =
    let sing, (++), (--) =
        Set.singleton,
        Set.union,
        Set.difference
    let rec fV = function
        | V x             -> sing x
        | App (t1,t2) -> fV t1 ++ fV t2
        | Lbd (v,t)     -> fV t -- sing v
        | _               -> Set.empty
    fV
```

- Den Term $A[x := B]$ erhält man aus dem Term A, indem man alle freien Vorkommen der Variablen x in A durch den Term B ersetzt.
- Eine Substitution $A[x := B]$ ist zulässig, wenn keine der freien Variablen von B durch die Substitution gebunden werden.

- Eine unzulässige Substitution

$$\lambda\, xy.(x\ y\ z)[z := x]$$

- Eine zulässige Substitution

$$\lambda\, xy.(x\ y\ z)[z := u]$$

Nachdem wir in den Übungen gesehen haben, wie wir stets zulässig substituieren können, wollen wir uns nun davon überzeugen, dass Rechnen im λ-Kalkül im wesentlichen durch Substitutionen erreicht werden kann.

Aufgabe

Was denken Sie was das Resultat ist wenn wir den Term

$$(((\lambda fgx.(f\ (g\ x))\ (add\ 3))\ (add\ 2))\ 0)$$

"ausrechnen". (Lösung siehe Wandtafel)

Wir haben gesehen, dass Terme "ausrechnen" im λ-Kalkül darauf beruht substitutionen vorzunehmen und damit Terme so lange zu vereinfachen, bis sie in einer möglichst einfachen Form vorliegen.

- Beim Vereinfachen eines Termes spricht man von einer "Konversion". Wir werden im folgenden vier Arten von Konversionen betrachten $(\alpha, \beta, \eta, \delta)$.
- Das Ziel der Konversionen, einen "nicht mehr zu vereinfachenden Term" nennt man eine Normalform. Wir werden die $\beta-$Normalform betrachten.

- Ziel der α-Konversion ist es "äquivalente" Terme wie $\lambda\, x.x$ und $\lambda\, y.y$ ineinander überführen zu können.
- Die α-Konversion formalisiert die Idee, dass die Bedeutung eines Terms nicht von den verwendeten Variablen sondern nur von seiner Struktur abhängt.

Definition

Die α-Konversion:

$$\lambda\, x.A \Rightarrow_\alpha \lambda\, y.A[x := y]$$

wenn y nicht in A vorkommt.

Beispiel:

$$\lambda\, fgx.(g\ f\ x) \Rightarrow_\alpha \lambda\, hgx.(g\ h\ x)$$

- Die β-Konversion formalisiert die Idee der Funktionsanwendung durch ersetzen von Werten durch entsprechende Funktionswerte.

Definition

Die β-Konversion:

$$(\lambda\, x.A\ B) \Rightarrow_\beta A[x := B]$$

wobei die Substitution zulässig sein muss.

Beispiel:

$$(\lambda\, x.(Add\ x\ x)\ z) \Rightarrow_\beta (Add\ z\ z)$$

- Die η-Konversion formalisiert die Idee, dass Funktionen, die dieselben Rückgabewerte produzieren gleich sind.

Definition

Die η-Konversion:

$$(\lambda\, x.A\, x) \Rightarrow_\eta A$$

wobei $x \notin FV(A)$ gelten muss und A keine Konstante ist, die nicht eine Funktion darstellt.

Beispiel:

$$\lambda\, x.(Add\ y\ x) \Rightarrow_\eta (Add\ y)$$

- In λ-Kalkülen mit Konstanten bestimmen δ-Konversionen die Bedeutung der Konstanten.
- Beispiel:

$$(\mathit{Add}\ 14)\ 3 \Rightarrow_\delta 17$$

$$\lambda f.\lambda x.(f\ (f\ x))\ (Add\ 3)\ 2$$
$$\Rightarrow_\beta \lambda x.((Add\ 3)\ ((Add\ 3)\ x))\ 2$$
$$\Rightarrow_\beta ((Add\ 3)\ ((Add\ 3)\ 2))$$
$$\Rightarrow_\delta ((Add\ 3)\ 5)$$
$$\Rightarrow_\delta 8$$

- Den Übergang von einem Term der Form $(\lambda\, x.A\ B)$ zu $A[x := B]$ mittels β-Konversion nennen wir eine β-Reduktion.
- Einen Term auf den wir eine β-Reduktion anwenden können nennen wir einen β Redex.
- Den Übergang von einem Term A in einen Term B durch δ-Koversion, wobei in B der Wert der entsprechenden Konstante eingesetzt wurde, nennen wir eine δ-Reduktion.
- Ein Term ist in β-Normalform, wenn er keinen β-Redex als Teilterm enthält und keine δ-Reduktion mehr möglich ist. (d.h. im Term ist keine β- oder δ-Reduktion mehr möglich).
- Ein Term A evaluiert zum Term B, wenn B in β-Normalform ist und eine endliche Sequenz von Reduktionen von A nach B existiert.

Die Anzahl der Redexe verringert sich mit einer Reduktion nicht notwendigerweise:

$$\lambda f.(f\ f\ f)\ (\lambda x.A)$$
$$\Rightarrow_\beta (\lambda x.A)\ (\lambda x.A)\ (\lambda x.A)$$

Nicht jeder Term kann zu einer β-Normalform reduziert werden:

$$\lambda\, x.(x\ x)\ \ \lambda\, x.(x\ x)$$

Die Reihenfolge in der Reduktionen angewendet werden können ist im Allgemeinen nicht eindeutig:

- Reduktion "von innen nach aussen":

$$
\begin{aligned}
&\lambda x.y\ (\lambda w.(w\ w)\ \lambda x.x) \\
&\Rightarrow_\beta \lambda x.y\ (\lambda x.x\ \lambda x.x) \\
&\Rightarrow_\beta \lambda x.y\ \lambda x.x \\
&\Rightarrow_\beta y
\end{aligned}
$$

- Reduktion "von Links nach Rechts":

$$
\begin{aligned}
&\lambda x.y\ (\lambda w.(w\ w)\ \lambda x.x) \\
&\Rightarrow_\beta y
\end{aligned}
$$

Evaluationsstrategien:

- "Normal order reduction" entspricht der Reduktionsstrategie "von Links nach Rechts"; der jeweils am weitesten links stehende Redex wird evaluiert. Lazy evaluation ist eine Variante dieser Strategie.
- "Applicative order reduction" entspricht der Reduktionsstrategie "von innen nach aussen"; der jeweils innerste Redex wird zuerst reduziert[1]. Strict evaluation ist eine Variante dieser Strategie.

[1] Die Argumente einer Funktion werden ausgewertet bevor die Funktion selbst ausgewertet wird.

- Wenn ein Term eine β-Normalform besitzt, dann wird diese immer durch "normal order reduction" gefunden.
- Es gibt Terme, die eine β-Normalform besitzen, die nicht mit "applicative order reduction" gefunden werden kann.
- Unabhängig von der gewählten Evaluationsstrategie, ist die (falls vorhanden) erhaltene β-Normalform bis auf α-Konversion eindeutig.

Ein Term, der mit "normal order reduction" nicht aber mit "applicative order reduction" reduziert werden kann:

- Applicative order:

$$
\begin{aligned}
&\lambda x.y\ (\lambda z.(z\ z\ z)\ \ \lambda z.(z\ z\ z))\\
&\Rightarrow_\beta \lambda x.y\ (\lambda z.(z\ z\ z)\ \ \lambda z.(z\ z\ z)\ \ \lambda z.(z\ z\ z))\\
&\Rightarrow_\beta \dots
\end{aligned}
$$

- Normal order reduction:

$$
\begin{aligned}
&\lambda x.y\ (\lambda z.(z\ z\ z)\ \ \lambda z.(z\ z\ z))\\
&\Rightarrow_\beta y
\end{aligned}
$$

- Der λ-Kalkül beinhaltet keine "expliziten" Konstrukte um rekursiv Funktionen zu deklarieren.
- Rekursion wird über Fixpunkte erreicht.
- Der Fixpunktkombinator Y (entspricht im wesentlichen `fix`) kann im untypisierten λ-Kalkül direkt als Term geschrieben werden:

$$Y :\equiv \lambda f.(\lambda x.(f\ (x\,x))\ \lambda x.(f\ (x\,x)))$$

Die Fixpunkteigenschaft $(Y\ g) = (g\ (Y\ g))$ kann einfach überprüft werden:

$$
\begin{aligned}
&(Y\ g) =_{Def.} \lambda f.(\lambda x.(f\ (x\ x))\ \lambda x.(f\ (x\ x)))\ g \\
&\Rightarrow_\beta (\lambda x.(g\ (x\ x))\ \lambda x.(g\ (x\ x))) \\
&\Rightarrow_\beta g\ (\lambda x.(g\ (x\ x))\ \lambda x.(g\ (x\ x))) \\
&\Rightarrow_\beta g\ (\lambda f.(\lambda x.(f\ (x\ x))\ \lambda x.(f\ (x\ x)))\ g) \\
&=_{Def}\ g\ (Y\ g)
\end{aligned}
$$

Weitere gängige Konstrukte und Datentypen der Programmierung lassen sich im λ-Kalkül simulieren:

- While Schleifen via Rekursion (mittels Fixpunkten).
- Natürliche Zahlen (durch “Church Numerale”).
- Paare und deren Projektionen (mittels expliziter Paarungsfunktion).
- Listen (via Paare).
- ...

Folgerung: Der λ-Kalkül ist Turing-vollständig.