Inhaltsverzeichnis

1 Geschichte und Einführung

1.1 Geschichte

1. Lambda Kalkül
   - Gleichwertig mit Turing Maschine (kann gleichen Probleme lösen)
   - Universale Turing Maschine (kann alle Turingmaschinen beschreiben)
   - Universale Lambda-Funktion gesucht
2. Fortran (Fortran list processing language)
3. Fortran M-Expressions (Idee wurde nie implementiert)
4. Lisp
   - List processing Alternative zu Turingmaschine: LISP Funktion eval
   - Notation: Programme und Daten können beide als Liste beschrieben werden

Abbildung 1: LISP EVAL vereinfacht

   - EVAL ist LISP Interpreter
5. Scheme
   - Prototyp für funktionale Programmiersprache
   - Versuch, LISP simpler umzusetzen
   - Optimized tail recursion (garantiert vom Compiler)

1.2 Tail call optimization

Tailcall Rekursion ist fast so effizient wie **goto** Begehle weil keine neuen Stackframes aufgebaut werden müssen.

1.2.1 Tail recursion

Rekursive Funktion f ist **endrekursiv**, wenn der rekursive Funktionsaufruf die **letzte Aktion zur Berechnung von f** ist.

1.2.2 Beispiele

```
// Kein tail call
int fact(int x){
    if(x == 0) return 1;
    else return fact(x - 1) * x;
}
```

```
// Tail call
int fact(int x, int accum=1){
    if(x == 0) return accum;
    else return fact(x - 1, x * accum);
}
```

Optimierung möglich, da x nicht als Zwischenergebnis gespeichert werden muss.

2 Closures

2.1 Static und Dynamic Scoping

- Static/Lexical Scoping:
  Struktur des Sourcecodes legt fest welche Variablen gemeint sind.
- Dynamic Scoping:
  Stack/Programmzustand legt Variablen fest.

2.2 Closure

Jede Scheme-Funktion hat eigene Umbebung für alle Variablen, die im body vorkommen.

Closure = Code + Umgebung

2.2.1 Beispiel

```
1  define (new-account)         ;; funktion ohne argumente
2      (let ((balance 0))      ;; lokale variable = 0
3      (lambda (x)             ;; Rückgabe = Funktion erhöht
           balance um x und gibt balance zurpck
4          (set! balance (+ balance x))
5          balance)))
6
7  (define ferien (new-account))
8  (ferien 10)             ;; -> 10
9  (ferien 10)             ;; -> 20
10
11 (define reisen (new-account))
12 (reisen 3)              ;; -> 3
13 (ferien 10)             ;; -> 30
```

3 Lambda Kalkül

3.1 Bestandteile

- **Term** *(Kodierte Werte und Funktionen)*
- **Variable**
- **Funktion**
- **λ-Abstraktion**

3.2 Terme

- Eine Variable ist ein Term
- Sind **t** und **s** Terme, so ist **ts** ein Term ***(wird auch Anwendung genannt)***.
- Ist **t** ein Term und **x** eine Variable, dann ist $\lambda x.t$ eine Abstraktion

3.3 Notation

- Linksklammerung: $xxx = (xx)x$ *(Rechtsklammerung muss explizit sein)*
- Abstraktion mit mehreren Variablen: $\lambda x.\lambda y.\lambda z.t = \lambda xyz.t$

4 Gleicheit

4.1 Gebundene Variablen

- Alle nicht freien Variablen
- können umbennant werden
  (Solange dadurch keine freien Variablen eingefangen werden)

4.1.1 Beispiel

```
1  inline int f(int x){
2    int i;
3    for(i = 0; i < 10; i++) x=x*x;
4    return x
5  }
6
7  int main(){
8    int x; int i; x=3; i=6;
9    f(x); f(i); f(5);
10   return 0;
11 }
```

4.1.2 Freie Variablen

Variable x in Term t is freie Variable wenn $x \in FV(t)$
Eine Variable ist element von FV wenn:

- t eine Variable x ist: $\mathbf{FV(t) = x}$
- t eine Anwendung vs ist: $\mathbf{FV(t) = FV(v) \cap FV(s)}$
- t eine Abstraktion $\lambda x.s$ ist: $\mathbf{FV(t) = FV(s) \backslash x}$

4.2 Einsetzen in Terme

Einsetzen eines Terms s in einen Term t für die Variable x: $\mathbf{t[x \leftarrow s]}$

4.2.1 Regeln

- t ist Variable
  1. $t = x$; $x[x \leftarrow s] = s$
  2. $t = y$; $y[x \leftarrow s] = y$ *nicht-x-Variable wird nicht ersetzt*
- t ist Anwendung pq
  1. $t = pq$; $pq[x \leftarrow s] = p[x \leftarrow s]q[x \leftarrow s]$
- t ist Abstraktion

1. $t = \lambda x.r$; $\lambda x.r[x \leftarrow s] = \lambda x.r$
2. $t = \lambda y.r$; $y \neq x$
   - $y \notin FV(s) \implies \lambda y.r[x \leftarrow s] = \lambda y.(r[x \leftarrow s])$
   - $y \in FV(s) \implies$ Umbenennung der gebundenen Variable: $\lambda y.r = \lambda z.r[y \leftarrow z] \implies (\lambda z.r[y \leftarrow z])[x \leftarrow s]$

4.2.2 Beispiele

- $x[x \leftarrow \lambda z.z] = \lambda z.z$
- $yy[x < -\lambda z.z] = yy$
- $(\lambda z.xy)[x \leftarrow y] = \lambda z.yy$
- $(\lambda z.xy)[x \leftarrow z] = \lambda p.(xy[z \leftarrow p])[x \leftarrow z] = \lambda p.zy$

4.3 α-Gleicheit

$\lambda x.t = \lambda y.(t[x \leftarrow y])$ *(falls y nicht in FV(t))*

4.4 β-Gleicheit

$(\lambda x.t)s = t[x \leftarrow s]$

4.5 η-Gleicheit

$(\lambda x.tx) = t$ *(falls x nicht in FV(t))*

5 Makros

5.1 Booleans und If

- true: λxy.x
- false: λxy.y
- if: λxyz.xyz
- not: λx.if x false true
- and: λxy.if x y false

5.2 Zahlen (Church Nummerals)

- 0 = λfx.x
- 1 = λfx.fx
- 2 = λfx.f(fx)
- n = $\lambda fx.f^n x$

5.2.1 Addition

λmn.λfx.mf(nfx)

5.2.2 Multiplikation

λmn.λfx.m(nf)x

5.3 Listen

- cons: λabc.if c a b
- car: λp.p true
- cdr: λp.p false

5.4 For loop

n f *(n = Church Numeral)*

6 Rekursion

6.1 Primitive Rekursion

Kann in endlichen Schritten gelöst werden:
if, for, listen, etc...

6.2 Allgemeine Rekursion

Nicht ohne Fixpunktoperator möglich

6.2.1 Fixpunktoperator (Y-Combinator)

- **Fixpunkt:** f = λx.if (< x 2) 1 (+ (f (- x 1)) (f (- x 2)))
- **Rekursion:** F = λfx.if (< x 2) 1 (+ (f (- x 1)) (f (- x 2)))
- f = F f *(f ist ein Fixpunkt von F)*

Für ein beliebiges **F** ist **YF** ein Fixpunkt: Y = λg.(λx.g(xx)) (λx.g(xx))

1. YF = λ(g.(λx.g(xx))(λx.g(xx))) F
2. (λx.F(xx))(λx.F(xx))
3. F((λx.F(xx)) (λx.F(xx)))
4. F((λg.(λx.g(xx)) (λx.g(xx))) F)

7 Lambda Umgebung

Ist **partielle Abbildung der variablen** in die Terme einer Definitionsmenge **(Domain)**

7.1 Spezielle Umgebungen

- εist die **leere Umgebung**
- Umgebungen mit genau einer Variablen sind **Bindungen**
  - Schreibweise: $x \to t$, $dom(\phi) = x$, $\phi(x) = t$
  - Komposition $\phi \circ \psi$ *(ϕund ψsind Umgebungen)*:
    - $dom(\phi \circ \psi) = dom(\phi) \cup dom(\psi)$
    - $$(\phi \circ \psi)(x) = \begin{cases} x \in dom(\phi) & \phi(x) \\ sonst & \psi(x) \end{cases}$$
    - dom(ϕ) ist endlich und besteht aus Bindungen ψ_i
      $\phi = \psi_0 \circ \psi_1 \circ \psi_2 \cdots \psi_n$

7.2 Auswertung

Die Auswertung einer Variablen **t** in ϕwird geschrieben $t \downarrow \phi$

7.2.1 Regeln

1. Variable x
   - $x \downarrow \phi = \phi(x)$
   - $x \downarrow \phi = x$ *(wenn $x \notin dom(\phi)$)*
2. Anwendung s auf r **(rs)**
   sei $p = r \downarrow \phi$ und $q = s \downarrow \phi$
   - wenn p die Form λx.u hat so ist $rs \downarrow \phi = u \downarrow [q \leftarrow x] \circ \phi$
   - Andernfalls: $rs \downarrow \phi = p \circ q$
3. Abstraktion λx.s
   $\lambda x.s \downarrow \phi = \lambda z.(s \downarrow (z \leftarrow x) \circ \phi)$
   Wobei $z \notin FV(s) \cup \bigcup_{y \in dom(\phi)} FV(\phi(y))$

7.2.2 Beispiele

```
(let ((a 1))
     (plus10 (let ((a 10))
                  (lambda (x) (+ a x))))
     (let ((y 5) (x 3))
          (plus10 (+ a y))))
```

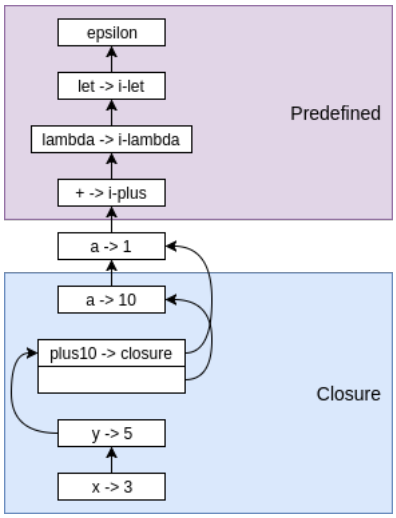

Abbildung 2: Lambda Bindings

8 λ-Term Auswertung

8.1 Beispiele: Umgebung

- $(\lambda x.t)s \downarrow \phi$
  - $s \downarrow \phi$
  - $t \downarrow (s \leftarrow x) \circ \phi$
  - $x \downarrow \phi = x$ falls $x \notin dom(\phi)$
- $(\lambda x.t) \downarrow \phi = \lambda x.(t \downarrow \phi)$ falls $x \notin \phi(y)$ für alle $y \in dom(y)$.
- $\lambda mnfx.mf(nfx)$ und $1 = \lambda gy.gy$

1. $((+1)1) \downarrow \varepsilon$
2. $1 \downarrow \varepsilon \rightarrow \lambda g.(\lambda y.gy) \downarrow \varepsilon$
3. $\lambda g.(\lambda y.gy) \downarrow \varepsilon$
4. $\lambda g.((\lambda y.gy) \downarrow \varepsilon)$
5. $\lambda g.\lambda y.(gy \downarrow \varepsilon) = \lambda g.\lambda y.(g \downarrow \varepsilon)(y \downarrow \varepsilon) = 1$
6. $(+1) \downarrow \varepsilon$
7. $\lambda m.(\lambda nfx.mf(nfx))1 \downarrow \varepsilon$
8. $\lambda nfx.(mf(nfx) \downarrow [1 \leftarrow m] \circ \varepsilon)$
9. $\lambda nfx.(m \downarrow [1 \leftarrow m] \circ \varepsilon)f(nfx)$
10. $\lambda nfx.1f(nfx)$
11. $(+1)1 \downarrow \varepsilon$