Erinnerung: Hintergrund & Grundlagen

- Auswertung von Ausdrücken
  Schlagwörter: applicative und normal order evaluation, eager und lazy evaluation, ...
  ...siehe Newsgruppenbeitrag vom 15.10.2003

  $\leadsto$ *erledigt.*

- λ-Kalkül
  ...formale Fundierung nahezu aller funktionalen Programmiersprachen

  $\leadsto$ *noch offen...*

Der λ-Kalkül...

...ausgezeichnet durch

- Einfachheit
  ...nur wenige syntaktische Konstrukte, einfache Semantik
- Ausdruckskraft
  ...Turing-mächtig, alle intuitiv berechenbaren Funktionen im λ-Kalkül ausdrückbar

$\leadsto$ Bindeglied zwischen funktionalen Hochsprachen und ihren maschinennahen Implementierungen.

Der λ-Kalkül: Syntax

Die Menge *Exp* der Ausdrücke des λ-Kalküls, kurz λ-Ausdrücke, ist definiert durch:

- Jeder Name (Identifier) ist in *Exp*. (Bsp: 1, n, $*$, fact, ...)
- *Applikation*: (f e) ist in *Exp*, wenn f und e in *Exp* sind; *Sprechweise*: Anwendung von f auf e.
  (Bsp.: fact 3, simple x y z (entspricht ((simple x) y) z), ...)
- *Abstraktion*: (λx.e) ist in *Exp*, wenn x ein Name und e aus *Exp* ist. *Sprechweise*: Funktionsabstraktion mit formalem Parameter x und Rumpf e.
  (Bsp.: λx.(x x), λx.λy.λz.(x * (y + z)), ...)

Konvention:
Überflüssige Klammern können weggelassen werden.
Bsp: x$*$y entspricht (($*$) x) y und e+z entspricht ((+) e) z

Freie und gebundene Variablen(vork.)

Die Menge der *frei* vorkommenden Variablen...

$$free(x) = \{x\}, \text{ wenn x ein Name ist}$$
$$free(f\ e) = free(f) \cup free(e)$$
$$free(\lambda x.e) = free(e) \setminus \{x\}$$

Umgekehrt...

Die Menge der *gebunden* vorkommenden Variablen...

$$bound(f\ e) = bound(f) \cup bound(e)$$
$$bound(\lambda x.e) = bound(e) \cup \{x\}$$

Es gilt... jedes Vorkommen (!) ist entweder gebunden oder frei.

(Syntaktische) Substitution

$$x[e/x] = e, \text{ wenn x ein Name ist}$$
$$y[e/x] = y, \text{ wenn y ein Name mit } x \neq y \text{ ist}$$
$$(f\ g)[e/x] = (f[e/x])\ (g[e/x])$$
$$(\lambda x.f)[e/x] = \lambda x.f$$
$$(\lambda y.f)[e/x] = \lambda y.(f[e/x]), \text{ wenn } x \neq y \text{ und } y \notin free(e)$$
$$(\lambda y.f)[e/x] = \lambda z.((f[z/y])[e/x]), \text{ wenn } x \neq y \text{ und } y \in free(e), \text{ wobei } x \neq z \text{ und } z \notin free(e) \cup free(f)$$

λ-Kalkül: Semantik

Zentral: λ-Konversionen...

- α-Konversion (Umbenennung formaler Parameter)
  $\lambda x.e \Leftrightarrow \lambda y.e[y/x]$, wobei $y \notin free(e)$
- β-Konversion (Funktionsanwendung)
  $(\lambda x.f)\ e \Leftrightarrow f[e/x]$
- η-Konversion (Elimination redundanter Funktion)
  $\lambda x.(e\ x) \Leftrightarrow e$, wobei $x \notin free(e)$

$\leadsto$ führen auf eine operationelle Semantik des λ-Kalküls.

Sprechweise: (Von links nach rechts) gerichtete Varianten der β- und η-Konversion heißen β- und η-Reduktion; ((von rechts nach links) gerichtete Variante der β-Konversion heißt β-Abstraktion.)

Intuition hinter den Konversionsregeln

Noch einmal zusammengefasst...

- α-Konversion... erlaubt die konsistente Umbenennung formaler Parameter von λ-Abstraktionen
- β-Konversion... erlaubt die Anwendung einer λ-Abstraktion auf ein Argument
  (Achtung: Gefahr von Bindungsfehlern! Abhilfe: α-Konversion!)
- η-Konversion... erlaubt die Elimination redundanter λ-Abstraktionen

Bsp.: $(\lambda x.\lambda y.x+y)\ (y*2) \Rightarrow \lambda y.(y*2)+y$
$\leadsto$ Bindungsfehler ("y wird eingefangen")

Beispiel für λ-Reduktion

$$\begin{array}{ll}
(\lambda x.\lambda y.x*y)\ ((\lambda x.\lambda y.x+y)\ 9\ 5)\ 3 & (\beta\text{-Reduktion}) \\
\Rightarrow\ (\lambda x.\lambda y.x*y)\ ((\lambda y.9+y)\ 5)\ 3 & (\beta\text{-Reduktion}) \\
\Rightarrow\ (\lambda y.((\lambda y.9+y)\ 5)*y)\ 3 & (\beta\text{-Reduktion}) \\
\Rightarrow\ (\lambda y.(9+5)*y)\ 3 & (\beta\text{-Reduktion}) \\
\Rightarrow\ (9+5)*3 & (\beta\text{- und } \eta\text{-Reduktion nicht anwendbar})
\end{array}$$

$\leadsto$ Weitere Regeln zur Reduktion primitiver Operationen in erweitertem λ-Kalkül (Auswertung arithmetischer Ausdrücke, bedingte Anweisungen, Listenoperationen, ...), sog. δ-Regeln.

$\leadsto$ solche Erweiterungen praktisch notwendig und einsichtig, aber kaum relevant in der Theorie.

Reduktionsfolgen und Normalformen

- Ein λ-Ausdruck ist in *Normalform*, wenn er durch β-Reduktion und η-Reduktion nicht weiter reduzierbar ist.
- Wenn ein λ-Ausdruck zu einem λ-Ausdruck in Normalform konvertierbar ist, dann führt jede terminierende Reduktion des λ-Ausdrucks zum selben λ-Ausdruck in Normalform (bis auf α-Konversion).
- Reduktionsfolgen: normal order (leftmost-outermost), applicative order (leftmost-innermost), ...
- Beachte: Nicht jeder λ-Ausdruck ist zu einem λ-Ausdruck in Normalform konvertierbar (Endlosrekursion).
  Bsp.: (1) $\lambda x.(x\ x)\ \lambda x.(x\ x) \Rightarrow \lambda x.(x\ x)\ \lambda x.(x\ x) \Rightarrow \ldots$
  (2) $(\lambda x.42)\ (\lambda x.(x\ x)\ \lambda x.(x\ x))$ (hat Normalform!)
- Durch Reduktionen im λ-Kalkül sind genau jene Funktionen berechenbar, die Turing-berechenbar sind.

Church-Rosser-Theoreme

Theorem 1
Wenn $e_1 \Leftrightarrow e_2$, dann gibt es einen λ-Ausdruck e mit $e_1 \Rightarrow^* e$ und $e_2 \Rightarrow^* e$
(Konfluenzeigenschaft, Diamanteigenschaft)

Theorem 2
Wenn $e_1 \Rightarrow^* e_2$ und e_2 in Normalform, dann gibt es eine normal order Reduktionsfolge von e_1 nach e_2
(Standardisierungstheorem)

Problem: Rekursion

Erinnerung...

fact n = if n == 1 then 1 else (n ∗ fact (n − 1))

oder alternativ:

fact = λn.if n == 1 then 1 else (n ∗ fact (n − 1))

“**Problem**”...

λ-Abstraktionen sind anonym.

Lösung: Der Y-Kombinator

Y-Kombinator: Y = λf.(λx.(f (x x)) λx.(f (x x)))

Es gilt: Für jeden λ-Ausdruck e ist (Y e) zu (e (Y e)) konvertierbar:

$$\begin{aligned} Y\ e &\Rightarrow \lambda x.(e\ (x\ x))\ \lambda x.(e\ (x\ x)) \\ &\Rightarrow e\ (\lambda x.(e\ (x\ x))\ \lambda x.(e\ (x\ x))) \\ &\Leftrightarrow e\ (Y\ e) \end{aligned}$$

Damit lässt sich Rekursion realisieren. Intuition:

f = ··· f ··· (rekursive Darstellung)
⇝ f = λf.(··· f ···) f (λ-Abstraktion)
⇝ f = Y λf.(··· f ···) (nicht-rekursive Darstellung)

Bsp.: fact = Y λf.(λn.if n == 1 then 1 else n ∗ f (n − 1))

fact 1 ⇒ ... ⇒ 1 (nachrechnen!)

Praktisch relevant: Typisierter λ-Kalkül

Jedem λ-Ausdruck ist ein Typ zugeordnet

Beispiele:
3 :: Integer
(∗) :: Integer −> Integer −> Integer
(λx.2 ∗ x) :: Integer −> Integer
(λx.2 ∗ x) 3 :: Integer

Einschränkung: Typen müssen konsistent sein (wohltypisiert)

Problem: Selbstanwendung im Y-Kombinator

⇝ Nicht endlich typisierbar

Abhilfe: explizite Rekursion zum Kalkül hinzufügen (mit der Reduktionsregel Y e ⇒ e (Y e)) (sinnvoll auch aus Effizienzgründen!)

Zurück zu Haskell...

- Haskell beruht auf typisiertem λ-Kalkül
- Übersetzer überprüft, ob alle Typen konsistent sind
- Programmierer kann Typdeklarationen angeben (Sicherheit), muss aber nicht (bequem, manchmal unerwartete Ergebnisse)
- fehlende Typinformation wird vom Übersetzer inferiert (berechnet)
- Rekursive Funktionen direkt verwendbar
  (daher kein Y-Kombinator notwendig)

λ-artige Funktionsnotation in Haskell...

...am Beispiel der Fakultätsfunktion:

```
fact :: Integer -> Integer
fact = \n -> (if n == 1 then 1 else (n * fact (n - 1)))
```

Mithin in Haskell: “\” statt “λ” und “->” statt “.”

Anekdote (vgl. P. Pepper [4]):
$(n.\widehat{n+1}) \leadsto (\wedge n.n+1) \leadsto (\lambda n.n+1) \leadsto$ \n->$n+1$

Vorschau auf die nächsten Vorlesungstermine...

- Di, 18.11.2003, 13-14 Uhr, Treitl-Hörsaal
- Di, 18.11.2003, 18-19 Uhr, Treitl-Hörsaal, 3. Zusatzvorlesung
- Ab Di, 25.11.2003, jeden Dienstag von 13-14 Uhr im Treitl-Hörsaal