Thema beim letzten Mal...

Hintergrund & Grundlagen

- Auswertung von Ausdrücken
  Schlagwörter: applicative und normal order evaluation, eager und lazy evaluation, ...
  $\leadsto$ *erledigt* (siehe 3. Vorlesungsteil vom 19.10.2004).

Heutiges Thema

- λ-Kalkül
  ...formale Fundierung nahezu aller funktionalen Programmiersprachen
  $\leadsto$ *...noch offen*. Mehr dazu in dieser Vorlesung.

Intuitive vs. formale Berechenbarkeit

Ausgangspunkt...

Intuitiv berechenbar ..."wenn es eine *irgendwie machbare effektive mechanische Methode* gibt, die zu jedem Argument aus dem Definitionsbereich nach endlich vielen Schritten den Funktionswert konstruiert und die für alle anderen Argumente entweder mit einem speziellen Fehlerwert oder nie abbricht".

Zentrale Frage...
Lässt sich der Begriff "intuitiver Berechenbarkeit" formal fassen?

Zur Beantwortung nötig...
Formale Berechnungsmodelle!

...d.h. *Explikationen* des Begriffs "intuitiver Berechenbarkeit".

Der λ-Kalkül... (1)

- ...ein spezielles formales Berechnungsmodell, wie viele andere auch, z.B.
  - allgemein rekursive Funktionen (Herbrand 1931, Gödel 1934, Kleene 1936)
  - Turing-Maschinen (Turing 1936)
  - μ-rekursive Funktionen (Kleene 1936)
  - Markov-Algorithmen (Markov 1951)
  - ...
- ...geht zurück auf Alonzo Church (1936)
- ...Berechnungen über Paaren, Listen, Bäumen, auch unendlichen, Funktionen höherer Ordnung einfach ausdrückbar
- ...in diesem Sinne "praxisnäher/realistischer" als andere formale Berechnungsmodelle

Der λ-Kalkül... (2)

Church'sche These
Eine Funktion ist genau dann intuitiv berechenbar, wenn sie λ-definierbar ist (d.h. im λ-Kalkül ausdrückbar ist).

Beweis? ...schlechterdings unmöglich!

Aber...

Der λ-Kalkül... (3)

Man hat bewiesen...

- Alle der obigen Berechnungsmodelle sind gleich mächtig.

Das kann als Hinweis darauf verstanden werden, dass alle der obigen Berechnungsmodelle den Begriff wahrscheinlich "gut" charakterisieren!

Aber: es schließt nicht aus, dass morgen ein mächtigeres formales Berechnungsmodell gefunden wird, das dann den Begriff der intuitiven Berechenbarkeit "besser" charakterisierte.

Präzedenzfall: Primitiv rekursive Funktionen

- ...bis Ende der 20er-Jahre als adäquate Charakterisierung intuitiver Berechenbarkeit akzeptiert (auch von Hilbert)
- ...tatsächlich jedoch: echt schwächeres Berechnungsmodell
- ...Beweis: Ackermann-Funktion ist berechenbar, aber nicht primitiv rekursiv (Ackermann 1928)

Die Ackermann-Funktion

..."berühmtberüchtigtes" Beispiel einer

- zweifellos (intuitiv) berechenbaren, aber nicht primitiv rekursiven Funktion!

```
ack :: (Integer,Integer) -> Integer
ack (m,n)
  | m == 0                 = n+1
  | (m > 0) && (n == 0)    = ack (m-1,1)
  | (m > 0) && (n /= 0)    = ack (m-1,ack(m,n-1))
```

$\leadsto$ ...hier in Haskell-Notation!

Der λ-Kalkül... (4)

...ausgezeichnet durch

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

Darüberhinaus...

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

Wir unterscheiden...

- *Reiner* λ-Kalkül
  ...reduziert auf das "absolut Notwendige"
  $\leadsto$ besonders bedeutsam in Untersuchungen zur Theorie der Berechenbarkeit
- *Angewandte* λ-Kalküle
  ...syntaktisch angereichert, praxisnäher

Reiner λ-Kalkül: Syntax

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

- Jeder *Name* (*Identifier*) ist in *Exp*.
  (Bsp: $a, b, c, \ldots, x, y, z, \ldots$)
- *Abstraktion*: Wenn x ein Name und e aus *Exp* ist, dann ist auch (λx. e) in *Exp*. *Sprechweise*: Funktionsabstraktion mit formalem Parameter x und Rumpf e.
  (Bsp.: (λx.(x x)), (λx.(λy.(λz.(x (y z))))), ...)
- *Applikation*: Wenn f und e in *Exp* sind, dann ist auch (f e) in *Exp*; *Sprechweisen*: Anwendung von f auf e. f heißt auch *Rator*, e auch *Rand*.
  (Bsp.: ((λx.(x x)) y), ...)

Alternativ...

...die Syntax in (modifizierter) Backus-Naur-Form (BNF):

e	$::=$		(λ-Ausdrücke)
	$::=$	x	(Namen (Identifikatoren))
	$::=$	$\lambda\, x.e$	(Abstraktion)
	$::=$	$e\ e$	(Applikation)
	$::=$	(e)	

Konventionen

- Überflüssige Klammern können weggelassen werden. Dabei gilt:
  - *Rechts*assoziativität für λ-Sequenzen in Abstraktionen
    Bsp.: – λx.λy.λz.(x(y z)) kurz für (λx.(λy.(λz.(x(y z))))),
    – λx. e kurz für (λx. e)
  - *Links*assoziativität für Applikationssequenzen
    Bsp.: – $e_1\ e_2\ e_3 \ldots e_n$ kurz für $(\ldots((e_1\ e_2)\ e_3)\ldots e_n)$,
    – $(e_1\ e_2)$ kurz für $e_1\ e_2$
- Der Rumpf einer λ-Abstraktion ist der längstmögliche dem Punkt folgende λ-Ausdruck
  Bsp.: – λx.e f entspricht λx.(e f), nicht (λx. e) f

Angewandte λ-Kalküle

Angewandte λ-Kalküle sind syntaktisch angereichert.

Beispielsweise...

- Auch Konstanten, Funktionsnamen oder "übliche" Operatoren können Namen (im weiteren Sinn) sein
  (Bsp: 1, 3.14, *true*, *false*, $+$, $*$, $-$, fact, simple, ...)
- Ausdrücke können...
  - komplexer sein
    (Bsp.: if e then e_1 else e_2 fi ... statt cond e e_1 e_2 für cond geeignete Funktion)
  - getypt sein
    (Bsp.: 1 : $I\!N$, $true : Boole$, ...)
- ...

λ-Ausdrücke sind dann beispielsweise auch...

- Applikationen: fact 3, simple x y z (entspricht ((simple x) y) z), ...)
- Abstraktionen: λx.(x + x), λx.λy.λz.(x * (y - z)), $2 + 3$,
  (λx.if odd x then x*2 else x div 2 fi) 42, ...)

In der Folge

...erlauben wir uns die Annehmlichkeit Ausdrücke, für die wir eine eingeführte Schreibweise haben (z.B. $n * fact\ (n-1)$) in dieser gewohnten Weise zu schreiben, auch wenn wir die folgenden Ergebnisse für den reinen λ-Kalkül formulieren.

Rechtfertigung...

- Resultate der theoretischen Informatik, insbesondere

  Alonzo Church. *The Calculi of Lambda-Conversion*. Annals of Mathematical Studies, Vol. 6, Princeton University Press, 1941

  ...zur Modellierung von ganzen Zahlen, Wahrheitswerten, etc. durch geeignete Ausdrücke des reinen λ-Kalküls

Freie und gebundene Variablen (1)

...in λ-Ausdrücken

Die Menge der *frei* vorkommenden Variablen...

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

Umgekehrt...

Die Menge der *gebunden* vorkommenden Variablen...

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

Beachte: gebunden ≠ nicht frei !
...sonst wäre etwa “x gebunden in y”

Freie und gebundene Variablen (2)

Betrachte: (λx. (x y)) x

- Gesamtausdruck
  - x kommt frei und gebunden in (λx. (x y)) x vor
  - y kommt frei in (λx. (x y)) x vor
- Teilausdrücke
  - x kommt gebunden in (λx. (x y)) vor und frei in (x y) und in x
  - y kommt frei in (λx. (x y)), (x y) und y vor

Gebunden vs. gebunden an...

Wir müssen unterscheiden...

- Eine *Variable* ist *gebunden*...
- Ein *Variablenvorkommen* ist *gebunden an*...

Gebunden und *gebunden an* ...unterschiedliche Konzepte!

Letzteres meint:

- Ein (definierendes oder angewandtes) Variablenvorkommen ist an ein definierendes Variablenvorkommen gebunden

Definition

- *Definierendes* V.vorkommen ...Vorkommen unmittelbar nach einem λ
- *Angewandtes* V.vorkommen ...jedes nicht definierende Vorkommen

Der λ-Kalkül: Vorwärts zur Semantik

Zentral sind folgende Begriffe...

- (Syntaktische) Substitution
- Konversionsregeln / Reduktionsregeln

Syntaktische Substitution

Erster zentraler Begriff... (*syntaktische*) *Substitution*

$$x[e/x] = e, \text{ wenn } x \text{ ein Name ist}$$
$$y[e/x] = y, \text{ wenn } y \text{ ein Name mit } x \neq y \text{ ist}$$

$$(f\ g)[e/x] = (f[e/x])\ (g[e/x])$$

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

(Syntaktische) Substitution

Einige Beispiele zur Illustration...

- $((x\ y)\ (y\ z))\ [a+b/y] = ((x\ (a+b))\ ((a+b)\ z))$
- $\lambda x.\ (x\ y)\ [a+b/y] = \lambda x.\ (x\ (a+b))$
- Aber: $\lambda x.\ (x\ y)\ [a+b/x] = \lambda x.\ (x\ y)$
- Achtung: $\lambda x.\ (x\ y)\ [x+b/y] \rightsquigarrow \lambda x.\ (x\ (x+b))$
  ...ohne Umbenennung *Bindungsfehler*!

  Deshalb: $\lambda x.\ (x\ y)\ [x+b/y] = \lambda z.\ ((x\ y)[z/x])\ [x+b/y]$
  $= \lambda z.\ (z\ y)\ [x+b/y]$
  $= \lambda z.\ (z\ (x+b))$
  ...dank Umbenennung kein Bindungsfehler!

Konversionsregeln

Zweiter zentraler Begriff: 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 $x \notin free(e)$

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

Sprechweisen

...im Zusammenhang mit Konversionsregeln

- Von links nach rechts gerichtete Anwendungen der β- und η-Konversion heißen β- und η-Reduktion.
- Von rechts nach links gerichtete Anwendungen der β-Konversion heißen 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

$(\lambda x.\lambda y.x*y)\ ((\lambda x.\lambda y.x+y)\ 9\ 5)\ 3$ (β-Reduktion)
$\Rightarrow (\lambda x.\lambda y.x*y)\ ((\lambda y.9+y)\ 5)\ 3$ (β-Reduktion)
$\Rightarrow (\lambda y.((\lambda y.9+y)\ 5)*y)\ 3$ (β-Reduktion)
$\Rightarrow (\lambda y.(9+5)*y)\ 3$ (β-Reduktion)
$\Rightarrow (9+5)*3$ (β- und η-Reduktion nicht anwendbar)

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

$\leadsto$ solche Erweiterungen sind praktisch notwendig und einsichtig, aber für die Theorie (der Berechenkarkeit) kaum relevant.

Reduktionsfolgen & Normalformen (1)

- Ein λ-Ausdruck ist in *Normalform*, wenn er durch β-Reduktion und η-Reduktion nicht weiter reduzierbar ist.
- (Praktisch relevante) Reduktionsstrategien
  - normal order (leftmost-outermost)
  - applicative order (leftmost-innermost)

Reduktionsfolgen & Normalformen (2)

- *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!)

Zentrale Resultate:

- Wenn ein λ-Ausdruck zu einem λ-Ausdruck in Normalform konvertierbar ist, dann führt jede terminierende Reduktion des λ-Ausdrucks zum (bis auf α-Konversion) selben λ-Ausdruck in Normalform (bis auf α-Konversion).
- Durch Reduktionen im λ-Kalkül sind genau jene Funktionen berechenbar, die Turing-, Markov-, ... berechenbar sind!

Church-Rosser-Theoreme

Seien e_1 und e_2 zwei λ-Ausdrücke...

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

Informell ...wenn eine Normalform ex., dann ist sie eindeutig (bis auf α-Konversion)!

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
(sog. *Standardisierungstheorem*)

Informell ...normal order Reduktion terminiert am häufigsten!

Semantik von λ-Ausdrücken

- λ-Ausdrücke in Normalform lassen sich (abgesehen von α-Konversionen) nicht weiter vereinfachen (reduzieren)
- Nach dem 1. Church-Rosser-Theorem ist die Normalform eines λ-Ausdrucks eindeutig bestimmt, wenn sie existiert (wieder abgesehen von α-Konversionen)

Das legt folgende Sichtweise nahe...

- Besitzt ein λ-Ausdruck eine Normalform, so ist dies sein Wert.
- *Umgekehrt*: Die *Semantik* (*Bedeutung*) eines λ-Ausdrucks ist seine Normalform, wenn sie existiert, ansonsten ist sie undefiniert.

Rekursion. Und wie sie behandelt wird...

Erinnerung...
fact n = if n == 0 then 1 else (n $*$ fact (n $-$ 1))

oder alternativ:
fact = λn.if n == 0 then 1 else (n $*$ fact (n $-$ 1))

"Problem"...
λ-Abstraktionen sind *anonym*.

Lösung: Der Y-Kombinator

Y-Kombinator: $Y = \lambda f.(\lambda x.(f\ (x\ x))\ \lambda x.(f\ (x\ x)))$

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

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

Mithilfe des Y-Kombinators lässt sich Rekursion realisieren.
Intuition:

$$
\begin{aligned}
& f = \cdots f \cdots \text{ (rekursive Darstellung)} \\
\rightsquigarrow\ & f = \lambda f.(\cdots f \cdots)\ f \ \ (\lambda\text{-Abstraktion}) \\
\rightsquigarrow\ & f = Y\ \lambda f.(\cdots f \cdots) \text{ (nicht-rekursive Darstellung)}
\end{aligned}
$$

Bemerkung:
λ-Terme ohne freie Variablen heißen *Kombinatoren*.

Zur Übung empfohlen

...Anwendung des Y-Kombinators

Betrachte...

$$fact = Y\ \lambda f.(\lambda n.\text{if } n == 0 \text{ then } 1 \text{ else } n * f\ (n-1))$$

Rechne nach...

$$fact\ 1 \Rightarrow\ \ldots \Rightarrow 1$$

Überprüfe dabei...

Der Y-Kombinator realisiert Rekursion
durch wiederholtes Kopieren

Praktisch relevant: Typisierte λ-Kalküle

Jedem λ-Ausdruck ist ein Typ zugeordnet

Beispiele:

$$
\begin{aligned}
3 &:: \text{Integer} \\
(*) &:: \text{Integer} -> \text{Integer} -> \text{Integer} \\
(\lambda x.2 * x) &:: \text{Integer} -> \text{Integer} \\
(\lambda x.2 * x)\ 3 &:: \text{Integer}
\end{aligned}
$$

Einschränkung: Typen müssen konsistent sein (wohltypisiert)

Problem jetzt: Selbstanwendung im Y-Kombinator

$\rightsquigarrow$ Y nicht endlich typisierbar!

Abhilfe: explizite Rekursion zum Kalkül hinzufügen mittels Hinzunahme der Reduktionsregel $Y\ e \Rightarrow e\ (Y\ e))$
...nebenbei: zweckmäßig auch aus Effizienzgründen!

Zurück zu Haskell...

- Haskell beruht auf typisiertem λ-Kalkül
- Übersetzer/Interpretierer 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 in Haskell kein Y-Kombinator notwendig)

Ergänzende und weiterführende Literaturhinweise

- A. Church. *The Calculi of Lambda-Conversion*. Annals of Mathematical Studies, Vol. 6, Princeton University Press, 1941.
- H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. (Revised Edn.), North Holland, 1984.

Zum Abschluss für heute

...λ-artige Funktionsnotation in Haskell

...am Beispiel der Fakultätsfunktion:

```
fact :: Integer -> Integer
fact = \n -> (if n == 0 then 1 else (n * fact (n - 1)))
```

Mithin in Haskell: "\\" statt "λ" und "->" statt "."

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