GOETHE UNIVERSITÄT FRANKFURT AM MAIN

www.uni-frankfurt.de

Einführung in die Funktionale Programmierung:

Funktionale Kernsprachen:

Einleitung & Der Lambda-Kalkül

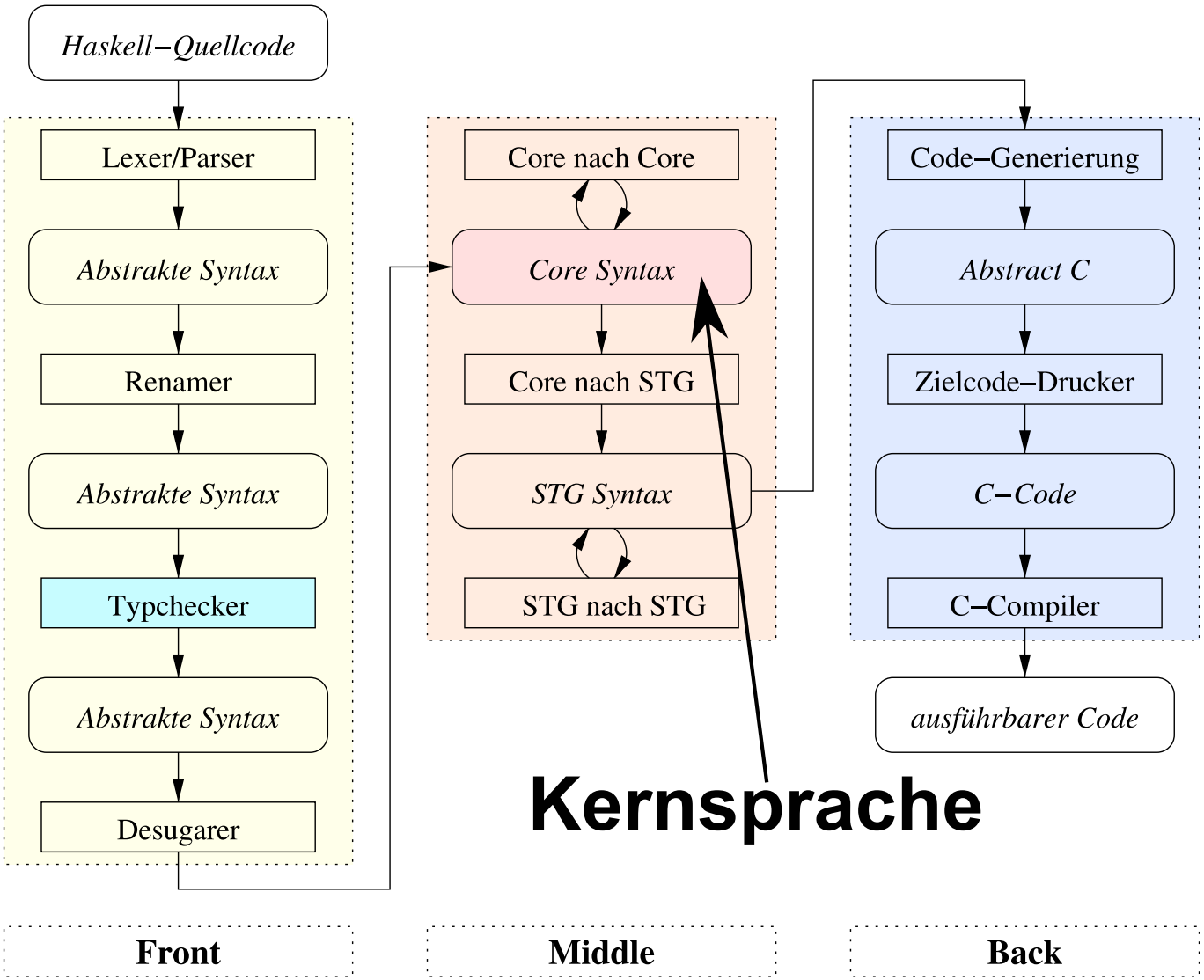
Prof. Dr. M. Schmidt-Schauß

WS 2025/26

Stand der Folien: 5. November 2025

Übersicht

Compilerphasen des GHC (schematisch)



Einleitung

- Wir betrachten zunächst den Lambda-Kalkül
- Er ist „Kernsprache“ fast aller (funktionalen) Programmiersprachen
- Allerdings oft zu minimalistisch,
  später: Erweiterte Lambda-Kalküle
- Kalkül: Syntax und Semantik
- Sprechweise: der Kalkül (da mathematisch)

Kalküle

Syntax

- Legt fest, welche Programme (Ausdrücke) gebildet werden dürfen
- Welche Konstrukte stellt der Kalkül zu Verfügung?

Semantik

- Legt die Bedeutung der Programme fest
- Gebiet der formalen Semantik kennt verschiedene Ansätze
  $\rightarrow$ kurzer Überblick auf den nächsten Folien

Ansätze zur Semantik

Axiomatische Semantik

- Beschreibung von Eigenschaften von Programmen mithilfe logischer Axiome und Schlussregeln
- Herleitung neuer Eigenschaften mit den Schlussregeln
- Prominentes Beispiel: Hoare-Logik, z.B. Hoare-Tripel $\{P\}S\{Q\}$:

  Vorbedingung P, Programm S, Nachbedingung Q

  Schlussregel z.B.:

$$\text{Sequenz: } \frac{\{P\}S\{Q\}, \{Q\}T\{R\}}{\{P\}S;T\{R\}}$$

- Erfasst i.a. nur einige Eigenschaften, nicht alle, von Programmen

Ansätze zur Semantik (2)

Denotationale Semantik

- Abbildung von Programmen in mathematische (Funktionen-)Räume durch Semantische Funktion
- Oft Verwendung von partiell geordneten Mengen (Domains)
- Z.B. $[\![\cdot]\!]$ als semantische Funktion:

$$[\![\texttt{if } a \texttt{ then } b \texttt{ else } c]\!] = \begin{cases} [\![b]\!], \text{ falls } [\![a]\!] = \texttt{True} \\ [\![c]\!], \text{ falls } [\![a]\!] = \texttt{False} \\ \bot, \text{ sonst} \end{cases}$$

- Ist eher mathematisch
- Schwierigkeit steigt mit dem Umfang der Syntax
- Nicht immer unmittelbar anwendbar (z.B. Nebenläufigkeit)

Ansätze zur Semantik (3)

Operationale Semantik

- definiert genau die Auswertung/Ausführung von Programmen
- definiert einen Interpreter
- Verschiedene Formalismen:
  - Zustandsübergangssysteme
  - Abstrakte Maschinen
  - Ersetzungssysteme
- Unterscheidung in small-step und big-step Semantiken
- Wir werden bevorzugt operationale (small-step) Semantiken verwenden

Der Lambda-Kalkül als Semantik für Programmiersprachen

Lambda-Kalkül als Semantik auch für imperative Programmiersprachen

- Lambda-Kalkül wird oft als semantischer Bereich für andere Programmiersprachen genutzt:

  Abbildung von Programmen/Ausdrücken in den Lambda-Kalkül.
- Grund ist, dass es eindeutig ist und sehr genau bekannt ist, wie man mit Namen, Funktionen, Funktionsanwendungen im Lambda-Kalkül umgeht.

Der Lambda-Kalkül

- Von Alonzo Church und Stephen Kleene in den 1930er Jahren eingeführt
- Idee und Ziel: Darstellungen der Berechnung von Funktionen

  $x \mapsto f(x)$ x ist Eingabe; $f(x)$ Ausgabe

  Mechanisches Rechenverfahren mit Funktionen
  (Alternative zu Turing-Maschine)
- Beispiele
- Der Lambda-Kalkül ist minimal: es geht NUR um Funktionen.
  Man definiert Funktionen;
  die Argumente sind ebenfalls Funktionen.

Der Lambda-Kalkül

- Wir betrachten den ungetypten Lambda-Kalkül.
- Wir geben die (eine) Auswertung als operationale Semantik an.
  Church, Kleene und auch andere Artikel / Bücher gehen meist axiomatisch vor (mit Gleichheitsaxiomen)
- Der ungetypte Lambda-Kalkül ist Turing-mächtig.
- Viele Ausdrücke des Lambda-Kalküls können in Haskell eingegeben werden, aber nur wenn sie Haskell-typisierbar sind.
- Haskell-Programm zur Auswertung von beliebigen Lambda-Ausdrücken:LExp.hs (siehe Webseite)

Berechnung im Lambda-Kalkül (informell)

vereinfachte Beispiele

$$(\lambda x.x)\ object \longrightarrow object$$

Berechnung im Lambda-Kalkül (informell)

vereinfachte Beispiele

$$(\lambda x.x)\ object \longrightarrow object$$
$$(\lambda\ x\ y.x)\ o1\ \ o2 \longrightarrow o1$$

Berechnung im Lambda-Kalkül (informell)

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

vereinfachte Beispiele

$$
\begin{array}{lcl}
(\lambda x.x)\ object & \longrightarrow & object \\
(\lambda\ x\ y.x)\ o1\ \ o2 & \longrightarrow & o1 \\
(\lambda\ x\ y.y)\ o1\ \ o2 & \longrightarrow & o2
\end{array}
$$

Berechnung im Lambda-Kalkül (informell)

vereinfachte Beispiele

$$
\begin{array}{lcl}
(\lambda x.x)\ object & \longrightarrow & object \\
(\lambda\ x\ y.x)\ o1\ \ o2 & \longrightarrow & o1 \\
(\lambda\ x\ y.y)\ o1\ \ o2 & \longrightarrow & o2 \\
(\lambda\ x\ y\ z.y)\ o1\ \ o2\ \ o3 & \longrightarrow & o2
\end{array}
$$

Berechnung im Lambda-Kalkül (informell)

vereinfachte Beispiele

$$
\begin{array}{lcl}
(\lambda x.x)\ object & \longrightarrow & object \\
(\lambda\ x\ y.x)\ o1\ \ o2 & \longrightarrow & o1 \\
(\lambda\ x\ y.y)\ o1\ \ o2 & \longrightarrow & o2 \\
(\lambda\ x\ y\ z.y)\ o1\ \ o2\ \ o3 & \longrightarrow & o2 \\
(\lambda\ f\ g\ x.f\ (g\ x))\ F\ \ G\ \ a & \longrightarrow & F\ (G\ \ a)
\end{array}
$$

Berechnung im Lambda-Kalkül (informell)

vereinfachte Beispiele

$(\lambda x.x)\ object$	$\longrightarrow$	$object$	Identität
$(\lambda\ x\ y.x)\ o1\ \ o2$	$\longrightarrow$	$o1$	Projektion links
$(\lambda\ x\ y.y)\ o1\ \ o2$	$\longrightarrow$	$o2$	Projektion rechts
$(\lambda\ x\ y\ z.y)\ o1\ \ o2\ \ o3$	$\longrightarrow$	$o2$	Projektion
$(\lambda\ f\ \ g\ \ x.f\ \ (g\ \ x))\ F\ \ G\ \ a$	$\longrightarrow$	$F\ \ (G\ \ a)$	Komposition von Funktionen

Syntax des Lambda-Kalküls

$\mathbf{Expr}$	$::=$	V	Variable (unendliche Menge)
	$\mid$	$\lambda V.\mathbf{Expr}$	Abstraktion
	$\mid$	$(\mathbf{Expr}\ \mathbf{Expr})$	Anwendung (Applikation)

- Abstraktionen sind anonyme Funktionen

  Z. B. $id(x) = x$ in Lambda-Notation: $\lambda x.x$

- Haskell: `\x -> s` als Programmcode für $\lambda x.s$
- $(s\ t)$ erlaubt die Anwendung von Funktionen auf Argumente

  s, t beliebige Ausdrücke $\Longrightarrow$ Lambda Kalkül ist higher-order

  Z. B. $(\lambda x.x)\ (\lambda x.x)$ (entspricht gerade $id(id)$)

Syntax des Lambda-Kalküls (2)

Assoziativitäten, Prioritäten und Abkürzungen

- Klammerregeln: $s\ r\ t$ entspricht $(s\ r)\ t$
- Priorität: Rumpf einer Abstraktion so groß wie möglich:

  $\lambda x.x\ y$ ist $\lambda x.(x\ y)$ und nicht $((\lambda x.x)\ y)$
- $\lambda x, y, z.s$ entspricht $\lambda x.\lambda y.\lambda z.s$ entspricht $\lambda x.(\lambda y.(\lambda z.s))$

Hierbei wird der Ausdruck von links nach rechts durchgegangen; (wie beim Parsen).

Z.B. $(x\ x\ z)\ (a\ b\ c)\ (d\ e\ f)$ entspricht geklammert:
$(\ ((x\ x)\ z)\ ((a\ b)\ c)\ )\ ((d\ e)\ f)$

Lambda-Kalkül Beispiele

- $\lambda x.x$ (Identitätsfunktion)
  Haskell-Eingabe: `\x -> x` z.B. `(\x->x) 1`

Lambda-Kalkül Beispiele

- $\lambda x.x$ (Identitätsfunktion)
  Haskell-Eingabe: `\x -> x` z.B. `(\x->x) 1`
- $\lambda x.\lambda y.x\ y$
  (Funktions-Anwendung des 1. auf das 2.Argument)

Lambda-Kalkül Beispiele

- $\lambda x.x$ (Identitätsfunktion)
  Haskell-Eingabe: `\x -> x` z.B. `(\x->x) 1`
- $\lambda x.\lambda y.x\ y$
  (Funktions-Anwendung des 1. auf das 2.Argument)
- $\lambda x.\lambda y.\lambda z.y$
  (Mittleres aus drei Argumenten ist das Resultat.)

Lambda-Kalkül Beispiele

- $\lambda x.x$ (Identitätsfunktion)
  Haskell-Eingabe: `\x -> x` z.B. `(\x->x) 1`
- $\lambda x.\lambda y.x\ y$
  (Funktions-Anwendung des 1. auf das 2.Argument)
- $\lambda x.\lambda y.\lambda z.y$
  (Mittleres aus drei Argumenten ist das Resultat.)
- $(\lambda x.\lambda y.y\ x)$
  (Das zweite Argument wird auf das erste angewendet)

Lambda-Kalkül Beispiele

- $\lambda x.x$ (Identitätsfunktion)
  Haskell-Eingabe: `\x -> x` z.B. `(\x->x) 1`
- $\lambda x.\lambda y.x\ y$
  (Funktions-Anwendung des 1. auf das 2.Argument)
- $\lambda x.\lambda y.\lambda z.y$
  (Mittleres aus drei Argumenten ist das Resultat.)
- $(\lambda x.\lambda y.y\ x)$
  (Das zweite Argument wird auf das erste angewendet)

Beispiele, was im Lambda Kalkül fehlt:

keine direkte Kodierung von Zahlen, Datenstrukturen.

kein unittelbares `if-then-else`

keine Iterationen, Drucken, ...

Bekannte Lambda-Ausdrücke

Oft vorkommende Ausdrücke (Kombinatoren)

$$
\begin{array}{rcll}
I & := & \lambda x.x & \text{Identität} \\
K & := & \lambda x.\lambda y.x & \text{Projektion auf Argument 1} \\
K2 & := & \lambda x.\lambda y.y & \text{Projektion auf Argument 2} \\
\Omega & := & (\lambda x.(x\ x))\ (\lambda x.(x\ x)) & \text{terminiert nicht} \\
Y & := & \lambda f.(\lambda x.(f\ (x\ x)))\ (\lambda x.(f\ (x\ x))) & \text{Fixpunkt Kombinator} \\
S & := & \lambda x.\lambda y.\lambda z.(x\ z)\ (y\ z) & S\text{-Kombinator zum SKI-calculus}
\end{array}
$$

$$
Y' := \lambda f.(\lambda x.(\lambda y.(f\ (x\ x\ y))))\ (\lambda x.(\lambda y.(f\ (x\ x\ y))))
$$

Fixpunkt Kombinator Variante für call-by-value

Beispiel zur Lambda-Notaion

Beispiel: algebraische Funktionen $f : \mathbb{Z} \to \mathbb{Z}$

Lambda-Kalkül mit Integers $\mathbb{Z}$ und einfacher Arithmetik

Operator F: soll Funktionen (bzw. deren Graph) um 2 nach unten verschieben.
Aktion: eine neue Funktion wird erzeugt

Beispiel zur Lambda-Notaion

Beispiel: algebraische Funktionen $f : \mathbb{Z} \to \mathbb{Z}$

Lambda-Kalkül mit Integers $\mathbb{Z}$ **und einfacher Arithmetik**

Operator F: soll Funktionen (bzw. deren Graph) um 2 nach unten verschieben.
Aktion: eine neue Funktion wird erzeugt

$F(f) := f'$, so dass $f'(x) = f(x) - 2$ für alle $x \in \mathbb{Z}$

Beispiel zur Lambda-Notaion

Beispiel: algebraische Funktionen $f : \mathbb{Z} \to \mathbb{Z}$

Lambda-Kalkül mit Integers $\mathbb{Z}$ **und einfacher Arithmetik**

Operator F: soll Funktionen (bzw. deren Graph) um 2 nach unten verschieben.
Aktion: eine neue Funktion wird erzeugt

$F(f) := f'$, so dass $f'(x) = f(x) - 2$ für alle $x \in \mathbb{Z}$

Mit Lambda-Notation formuliert:

(wenn man Zahlen und arithmetische Operatoren erlaubt)

$F := \lambda f.\lambda x.(f\ x) - 2$

F angewendet auf die "Funktion $f = \lambda y.y * y$":
$F\ (\lambda y.y * y)$ ergibt: $\lambda x.(f\ x) - 2 = \lambda x.x * x - 2$

Syntax und Semantik

Notation	(Syntax)	Wie schreibt man es hin?
Bedeutung	(Semantik)	Was beschreibt man?

Begriffe im Lambda-Kalkül (Syntax)

Freie und gebundene Variablen:
Durch λx ist x im Rumpf s von $\lambda x.s$ gebunden.
Kommt x in t vor, so

- ist das Vorkommen frei, wenn kein λx darüber steht
- andernfalls ist das Vorkommen gebunden

Beispiel:

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

- x kommt je einmal gebunden und frei vor
- y kommt gebunden vor
- z kommt frei vor

Begriffe im Lambda-Kalkül

Menge der freien und gebundenen Variablen

$FV(t)$: Freie Variablen von t

$$
\begin{aligned}
FV(x) &= x \\
FV(\lambda x.s) &= FV(s) \setminus \{x\} \\
FV(s\ t) &= FV(s) \cup FV(t)
\end{aligned}
$$

$BV(t)$: Gebundene Var. von t

$$
\begin{aligned}
BV(x) &= \emptyset \\
BV(\lambda x.s) &= BV(s) \cup \{x\} \\
BV(s\ t) &= BV(s) \cup BV(t)
\end{aligned}
$$

Begriffe im Lambda-Kalkül

Menge der freien und gebundenen Variablen

$FV(t)$: Freie Variablen von t

$$\begin{aligned} FV(x) &= x \\ FV(\lambda x.s) &= FV(s) \setminus \{x\} \\ FV(s\ t) &= FV(s) \cup FV(t) \end{aligned}$$

$BV(t)$: Gebundene Var. von t

$$\begin{aligned} BV(x) &= \emptyset \\ BV(\lambda x.s) &= BV(s) \cup \{x\} \\ BV(s\ t) &= BV(s) \cup BV(t) \end{aligned}$$

Wenn $FV(t) = \emptyset$, dann sagt man:

t ist geschlossen bzw. t ist ein Programm

Andernfalls: t ist ein offener Ausdruck

Z.B. $BV(\lambda x.(x\ (\lambda z.(y\ z)))) = \{x, z\}$

$FV(\lambda x.(x\ (\lambda z.(y\ z)))) = \{y\}$

Motivation zu Substitution

Anwendung einer Funktion auf ein Argument!

$(\lambda x.s)\ \ a$: Idee der Berechnung:
Resultat s', wobei:
s' ist s wobei jedes x in s durch a ersetzt wird.
(i.a. richtig; Sonderfälle brauchen mehr Erklärungen)

Beispiel

$((\lambda x.(\lambda y.x))\ \ id)$ ist $(\lambda y.id)$

Motivation zu Substitution

Anwendung einer Funktion auf ein Argument!

$(\lambda x.s)\ \ a$: Idee der Berechnung:
Resultat s', wobei:
s' ist s wobei jedes x in s durch a ersetzt wird.
(i.a. richtig; Sonderfälle brauchen mehr Erklärungen)

Beispiel

$((\lambda x.(\lambda y.x))\ \ id)$ ist $(\lambda y.id)$

Vorsicht:

$((\lambda x.(\lambda y.x))\ \ y)$ ergibt $(\lambda y.y)$?

Motivation zu Substitution

Anwendung einer Funktion auf ein Argument!

$(\lambda x.s)\ \ a$: Idee der Berechnung:
Resultat s', wobei:
s' ist s wobei jedes x in s durch a ersetzt wird.
(i.a. richtig; Sonderfälle brauchen mehr Erklärungen)

Beispiel

$((\lambda x.(\lambda y.x))\ \ id)$ ist $(\lambda y.id)$

Vorsicht:

$((\lambda x.(\lambda y.x))\ \ y)$ ergibt $(\lambda y.y)$? NEIN

Motivation zu Substitution

Anwendung einer Funktion auf ein Argument!

$(\lambda x.s)\ \ a$: Idee der Berechnung:
Resultat s', wobei:
s' ist s wobei jedes x in s durch a ersetzt wird.
(i.a. richtig; Sonderfälle brauchen mehr Erklärungen)

Beispiel

$((\lambda x.(\lambda y.x))\ \ id)$ ist $(\lambda y.id)$

Vorsicht:

$((\lambda x.(\lambda y.x))\ \ y)$ ergibt $(\lambda y.y)$? NEIN

Richtig ist:

$((\lambda x.(\lambda y.x))\ \ y)$ ergibt $(\lambda z.y)$ wobei $z \neq y$

Substitution

$s[t/x]$ = Idee: ersetze alle freien Vorkommen von x in s durch t

Formale Definition (genauer!)

$$
\begin{array}{lcl}
x[t/x] & = & t \\
y[t/x] & = & y, \text{ falls } x \neq y \\
(\lambda y.s)[t/x] & = & \begin{cases} \lambda y.(s[t/x]) & \text{falls } x \neq y \\ \lambda y.s & \text{falls } x = y \end{cases} \\
(s_1\ s_2)[t/x] & = & (s_1[t/x]\ s_2[t/x])
\end{array}
$$

Z.B. $(\lambda x.z\ x)[(\lambda y.y)/z] = (\lambda x.((\lambda y.y)\ x))$

Bedingung: $FV(t) \cap BV(s) = \emptyset$

Substitution

$s[t/x]$ = Idee: ersetze alle freien Vorkommen von x in s durch t

Wichtig ist: Vermeiden von nicht intendierten Bindungen.

Verfahren das es richtg macht:

1. Von innen her in jeder Abstraktion $\lambda x.t$ als Teilterm von s:
   (i) wähle einen komplett neuen Variablennamen z;
   (ii) ersetze alle Vorkommen von x in t durch z: ergibt t'
   (iii) ersetze dann $\lambda x.t$ durch $\lambda z.t'$.
2. Dann die Substitution ganz normal durchführen

Zur Sicherheit: Danach nochmal genauso umbenennen.

Kontexte

- Kontext = Ausdruck, der an einer Position ein Loch $[\cdot]$ anstelle eines Unterausdrucks hat

Grammatik für Kontexte:

$$\mathbf{C} = [\cdot] \mid \lambda V.\mathbf{C} \mid (\mathbf{C}\ \mathbf{Expr}) \mid (\mathbf{Expr}\ \mathbf{C})$$

- Sei C ein Kontext und s ein Ausdruck:

  $C[s]$ = Ausdruck, in dem das Loch in C durch s ersetzt wird
- Beispiel: $C = ([\cdot]\ (\lambda x.x))$, dann: $C[\lambda y.y] = ((\lambda y.y)\ (\lambda x.x))$.
- Das Einsetzen in Kontexte darf/kann freie Variablen einfangen:
  z.B. sei $C = (\lambda x.[\cdot])$, dann $C[\lambda y.x] = (\lambda x.\lambda y.x)$

Alpha-Äquivalenz

Alpha-Umbenennungsschritt

$$C[\lambda x.s] \xrightarrow{\alpha} C[\lambda y.s[y/x]] \text{ falls } y \notin BV(C[\lambda x.s]) \cup FV(C[\lambda x.s])$$

Alpha-Äquivalenz

$=_\alpha$ ist die reflexiv-transitive Hülle von $\xrightarrow{\alpha}$

- Wir betrachten α-äquivalente Ausdrücke als inhaltlich gleich.
- z.B. $\lambda x.x =_\alpha \lambda y.y$
- Syntaktische Konvention:
  Distinct Variable Convention: Alle gebundenen Variablen sind verschieden und gebundene Variablen sind verschieden von freien Variablen.
- Es gibt immer eine Folge von α-Umbenennungen, so dass die DVC danach erfüllt ist.

Beispiel zur DVC und α-Umbenennung

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

Beispiel zur DVC und α-Umbenennung

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

$\Longrightarrow$ erfüllt die DVC nicht.

Beispiel zur DVC und α-Umbenennung

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

$\Longrightarrow$ erfüllt die DVC nicht.

$$\begin{array}{rl} & (y\ (\lambda y.((\lambda x.(x\ x))\ (x\ y)))) \\ \xrightarrow{\alpha} & (y\ (\lambda y_1.((\lambda x.(x\ x))\ (x\ y_1)))) \\ \xrightarrow{\alpha} & (y\ (\lambda y_1.((\lambda x_1.(x_1\ x_1))\ (x\ y_1)))) \end{array}$$

$(y\ (\lambda y_1.((\lambda x_1.(x_1\ x_1))\ (x\ y_1))))$ erfüllt die DVC

Vorführung LEXP.hs


```
-- run erwartet den Ausdruck als String und reduziert alle Redexe
run str = printExp $ reduce (parse str)

-- runNO erwartet den Ausdruck als String und reduziert in Normalordnung
runNO str = printExp $ reduceNO (parse str)

-- runAO erwartet den Ausdruck als String und reduziert in Anwendungsordnung
runAO str = printExp $ reduceAO (parse str)

-- dvc erwartet den Ausdruck als String und pr"uft, ob der Ausdruck die
-- Distinct Variable Convention erf"ullt
dvc str = dvcexp (parse str)

-- fv erwartet den Ausdruck als String und berechnet das Paar: (gebundene Variablen, freie Variablen)
fv str = fvexp (parse str)

-- alphaEqual s t erwartet zwei Ausdr"ucke und testet, ob diese alpha-equivalent sind

alphaEqual s t  = alphaeq s t
-- wobei.. ..       alphaeq s t = alphaeqexp (parse s) (parse t)

--  dvcTrans  erwartet den Ausdruck als String und berechnet einen
--    alpha-aequivalenten der die dvc erfuellt.
dvcTrans s
```

Substitution: Nochmal genauer

Richtige Definition der Einsetzung löst das Problem

$s[t/x]$ = ersetze alle freien Vorkommen von x in s durch t,
falls $FV(t) \cap BV(s) = \emptyset$.

= $s'[t/x]$, sonst;
wobei s' aus s durch α-Umbenennung entsteht,
so dass $FV(t) \cap BV(s') = \emptyset$.

Wenn $((\lambda x.s)\ t)$ die DVC erfüllt,
dann sind die Bedingungen von Fall 1 erfüllt.

Operationale Semantik - Beta-Reduktion

Beta-Reduktion

$$(\beta) \qquad (\lambda x.s)\ t \to s[t/x]$$

Wenn $e_1 \xrightarrow{\beta} e_2$, dann sagt man e_1 reduziert unmittelbar zu e_2.

Operationale Semantik - Beta-Reduktion

Beta-Reduktion

$$(\beta) \qquad (\lambda x.s)\ t \to s[t/x]$$

Wenn $e_1 \xrightarrow{\beta} e_2$, dann sagt man e_1 reduziert unmittelbar zu e_2.

Beispiele:

$$(\lambda x.\underbrace{x}_{s})\ \underbrace{(\lambda y.y)}_{t} \xrightarrow{\beta} x[(\lambda y.y)/x] = \lambda y.y$$

$$(\lambda y.\underbrace{y\ y\ y}_{s})\ \underbrace{(x\ z)}_{t} \xrightarrow{\beta} (y\ y\ y)[(x\ z)/y] = (x\ z)\ (x\ z)\ (x\ z)$$

Beta-Reduktion: Umbenennungen

Damit die DVC nach einer β-Reduktion gilt, muss man (manchmal) umbenennen:

$$(\lambda x.(x\ x))\ (\lambda y.y) \xrightarrow{\beta} (\lambda y.y)\ (\lambda y.y) =_\alpha (\lambda y_1.y_1)\ (\lambda y_2.y_2)$$

Das wird bei mehreren Reduktionen hintereinander immer gemacht (aber nicht mehr erwähnt)

! In compilierten Programmen ist diese Operation nicht notwendig

De Bruijn Indizes

Das ist eine alternative Methode um Umbenennungen während der zu vermeiden.
De Bruijn Indizes (natürliche Zahlen) representieren die Tiefe der Schachtelung.
Die Methode Lambda-Ausdrücke zu reduzieren ist in etwa vergleichbar
mit einer Stackmaschine zur Reduktion
von Lambda Ausdrücken statt textueller Reduktion.

Operationale Semantik

- Für die Festlegung der operationalen Semantik
  (D.h. der Auswertung von Ausdrücken)
  des Lambda-Kalküls muss man noch definieren,
  wo in einem Ausdruck (an welcher Stelle im Syntaxbaum)
  die β-Reduktion angewendet wird
- Betrachte $((\lambda x.xx)((\lambda y.y)(\lambda z.z)))$.

  Zwei Möglichkeiten:
- $\underline{((\lambda x.xx)((\lambda y.y)(\lambda z.z)))} \rightarrow ((\lambda y.y)(\lambda z.z))\ ((\lambda y.y)(\lambda z.z))$
  oder
- $((\lambda x.xx)\underline{((\lambda y.y)(\lambda z.z))}) \rightarrow ((\lambda x.xx)(\lambda z.z))$.

Normalordnungsreduktion (wie in Haskell)

- Sprechweisen: Normalordnung, **call-by-name**, **nicht-strikt**, **lazy**
- Grob: Definitionseinsetzung ohne Argumentauswertung

Normalordnungsreduktion (wie in Haskell)

- Sprechweisen: Normalordnung, **call-by-name**, **nicht-strikt**, **lazy**
- Grob: Definitionseinsetzung ohne Argumentauswertung

Definition

- Reduktionskontexte $\mathbf{R} ::= [\cdot] \mid (\mathbf{R}\ \mathbf{Expr})$
- $\xrightarrow{no}$: Wenn $s \xrightarrow{\beta} t$, dann ist

$$R[s] \xrightarrow{no} R[t]$$

eine Normalordnungsreduktion

Normalordnungsreduktion (wie in Haskell)

- Sprechweisen: Normalordnung, **call-by-name**, **nicht-strikt**, **lazy**
- Grob: Definitionseinsetzung ohne Argumentauswertung

Definition

- Reduktionskontexte $\mathbf{R} ::= [\cdot] \mid (\mathbf{R}\ \mathbf{Expr})$
- $\xrightarrow{no}$: Wenn $s \xrightarrow{\beta} t$, dann ist

$$R[s] \xrightarrow{no} R[t]$$

eine Normalordnungsreduktion

Beispiel:

$$\begin{array}{rl} & ((\lambda x.(x\ x))\ (\lambda y.y))\ ((\lambda w.w)\ (\lambda z.(z\ z))) \\ \xrightarrow{\beta} & (x\ x)[(\lambda y.y)/x]\ ((\lambda w.w)\ (\lambda z.(z\ z))) \\ = & ((\lambda y.y)\ (\lambda y.y))\ ((\lambda w.w)\ (\lambda z.(z\ z))) \end{array}$$

$$R = ([\cdot]\ ((\lambda w.w)\ (\lambda z.(z\ z))))$$

Reduktionskontexte: Beispiele

Zur Erinnerung: $\mathbf{R} ::= [\cdot] \mid (\mathbf{R}\ \mathbf{Expr})$

$$\text{Sei } s = ((\lambda w.w)\ (\lambda y.y))\ ((\lambda z.(\lambda x.x)\ z)\ u)$$

Alle „Reduktionskontexte für s“, d.h. R mit $R[t] = s$ für irgendein t, sind:

- $R = [\cdot]$, Term t ist s selbst,
  für s ist aber keine β-Reduktion möglich

Reduktionskontexte: Beispiele

Zur Erinnerung: $\mathbf{R} ::= [\cdot] \mid (\mathbf{R}\ \mathbf{Expr})$

$$\text{Sei } s = ((\lambda w.w)\ (\lambda y.y))\ ((\lambda z.(\lambda x.x)\ z)\ u)$$

Alle „Reduktionskontexte für s“, d.h. R mit $R[t] = s$ für irgendein t, sind:

- $R = [\cdot]$, Term t ist s selbst,
  für s ist aber keine β-Reduktion möglich
- $R = ([\cdot]\ ((\lambda z.(\lambda x.x)\ z)\ u))$,
  Term t ist $((\lambda w.w)\ (\lambda y.y))$

  Reduktion möglich: $((\lambda w.w)\ (\lambda y.y)) \xrightarrow{\beta} (\lambda y.y)$.
  $s = R[((\lambda w.w)\ (\lambda y.y))] \xrightarrow{no} R[\lambda y.y] = ((\lambda y.y)\ ((\lambda z.(\lambda x.x)\ z)\ u))$

Reduktionskontexte: Beispiele

Zur Erinnerung: $\mathbf{R} ::= [\cdot] \mid (\mathbf{R}\ \mathbf{Expr})$

$$\text{Sei } s = ((\lambda w.w)\ (\lambda y.y))\ ((\lambda z.(\lambda x.x)\ z)\ u)$$

Alle „Reduktionskontexte für s“, d.h. R mit $R[t] = s$ für irgendein t, sind:

- $R = [\cdot]$, Term t ist s selbst,
  für s ist aber keine β-Reduktion möglich
- $R = ([\cdot]\ ((\lambda z.(\lambda x.x)\ z)\ u))$,
  Term t ist $((\lambda w.w)\ (\lambda y.y))$
  Reduktion möglich: $((\lambda w.w)\ (\lambda y.y)) \xrightarrow{\beta} (\lambda y.y)$.
  $s = R[((\lambda w.w)\ (\lambda y.y))] \xrightarrow{no} R[\lambda y.y] = ((\lambda y.y)\ ((\lambda z.(\lambda x.x)\ z)\ u))$
- $R = ([\cdot]\ (\lambda y.y))\ ((\lambda z.(\lambda x.x)\ z)\ u)$,
  Term t ist $(\lambda w.w)$,
  für $(\lambda w.w)$ ist keine β-Reduktion möglich.

Redexsuche: Markierungsalgorithmus

- s ein Ausdruck.
- Start: $s^\star$
- Verschiebe-Regel

$$(s_1\ s_2)^\star \Rightarrow (s_1^\star\ s_2)$$

so oft anwenden wie möglich.

Redexsuche: Markierungsalgorithmus

- s ein Ausdruck.
- Start: $s^\star$
- Verschiebe-Regel

$$(s_1\ s_2)^\star \Rightarrow (s_1^\star\ s_2)$$

  so oft anwenden wie möglich.
- Beispiel 1: $(((\lambda x.x)\ (\lambda y.(y\ y)))\ (\lambda z.z))^\star$

Redexsuche: Markierungsalgorithmus

- s ein Ausdruck.
- Start: $s^\star$
- Verschiebe-Regel

$$(s_1\ s_2)^\star \Rightarrow (s_1^\star\ s_2)$$

  so oft anwenden wie möglich.
- Beispiel 1: $(((\lambda x.x)\ (\lambda y.(y\ y)))^\star\ (\lambda z.z))$

Redexsuche: Markierungsalgorithmus

- s ein Ausdruck.
- Start: $s^\star$
- Verschiebe-Regel

$$(s_1\ s_2)^\star \Rightarrow (s_1^\star\ s_2)$$

  so oft anwenden wie möglich.
- Beispiel 1: $(((\lambda x.x)^\star\ (\lambda y.(y\ y)))\ (\lambda z.z))$

Redexsuche: Markierungsalgorithmus

- s ein Ausdruck.
- Start: $s^\star$
- Verschiebe-Regel

$$(s_1\ s_2)^\star \Rightarrow (s_1^\star\ s_2)$$

  so oft anwenden wie möglich.
- Beispiel 1: $(((\lambda x.x)^\star\ (\lambda y.(y\ y)))\ (\lambda z.z))$
- Beispiel 2: $(((y\ z)\ ((\lambda w.w)(\lambda x.x)))(\lambda u.u))^\star$

Redexsuche: Markierungsalgorithmus

- s ein Ausdruck.
- Start: $s^\star$
- Verschiebe-Regel

$$(s_1\ s_2)^\star \Rightarrow (s_1^\star\ s_2)$$

  so oft anwenden wie möglich.
- Beispiel 1: $(((\lambda x.x)^\star\ (\lambda y.(y\ y)))\ (\lambda z.z))$
- Beispiel 2: $(((y\ z)\ ((\lambda w.w)(\lambda x.x)))^\star(\lambda u.u))$

Redexsuche: Markierungsalgorithmus

- s ein Ausdruck.
- Start: $s^\star$
- Verschiebe-Regel

$$(s_1\ s_2)^\star \Rightarrow (s_1^\star\ s_2)$$

  so oft anwenden wie möglich.
- Beispiel 1: $(((\lambda x.x)^\star\ (\lambda y.(y\ y)))\ (\lambda z.z))$
- Beispiel 2: $(((y\ z)^\star\ ((\lambda w.w)(\lambda x.x)))(\lambda u.u))$

Redexsuche: Markierungsalgorithmus

- s ein Ausdruck.
- Start: $s^\star$
- Verschiebe-Regel

$$(s_1\ s_2)^\star \Rightarrow (s_1^\star\ s_2)$$

  so oft anwenden wie möglich.
- Beispiel 1: $(((\lambda x.x)^\star\ (\lambda y.(y\ y)))\ (\lambda z.z))$
- Beispiel 2: $(((y^\star\ z)\ ((\lambda w.w)(\lambda x.x)))(\lambda u.u))$

Redexsuche: Markierungsalgorithmus

- s ein Ausdruck.
- Start: $s^\star$
- Verschiebe-Regel

$$(s_1\ s_2)^\star \Rightarrow (s_1^\star\ s_2)$$

  so oft anwenden wie möglich.
- Beispiel 1: $(((\lambda x.x)^\star\ (\lambda y.(y\ y)))\ (\lambda z.z))$
- Beispiel 2: $(((y^\star\ z)\ ((\lambda w.w)(\lambda x.x)))(\lambda u.u))$

Allgemein: $(s_1\ s_2\ \ldots\ s_n)^\star$ hat das Ergebnis $(s_1^\star\ s_2\ \ldots\ s_n)$, wobei s_1 keine Anwendung

Redexsuche: Markierungsalgorithmus

- s ein Ausdruck.
- Start: $s^\star$
- Verschiebe-Regel

$$(s_1\ s_2)^\star \Rightarrow (s_1^\star\ s_2)$$

  so oft anwenden wie möglich.
- Beispiel 1: $(((\lambda x.x)^\star\ (\lambda y.(y\ y)))\ (\lambda z.z))$
- Beispiel 2: $(((y^\star\ z)\ ((\lambda w.w)(\lambda x.x)))(\lambda u.u))$

Allgemein: $(s_1\ s_2\ \ldots\ s_n)^\star$ hat das Ergebnis $(s_1^\star\ s_2\ \ldots\ s_n)$, wobei s_1 keine Anwendung

Falls $s_1 = \lambda x.t$ und $n \geq 2$ dann reduziere:

$$((\lambda x.t)\ s_2\ s_3\ \ldots s_n) \xrightarrow{no} (t[s_2/x]\ s_3\ \ldots s_n)$$

Beispiel: Suche nch Redex

$$(((\lambda x.\lambda y.x)((\lambda w.w)(\lambda z.z)))(\lambda u.u))^{\star}$$

Beispiel: Suche nch Redex

$$
\begin{array}{ll}
 & (((\lambda x.\lambda y.x)((\lambda w.w)(\lambda z.z)))(\lambda u.u))^{\star} \\
\Rightarrow & (((\lambda x.\lambda y.x)((\lambda w.w)(\lambda z.z)))^{\star}(\lambda u.u)) \\
\Rightarrow & (((\lambda x.\lambda y.x)^{\star}((\lambda w.w)(\lambda z.z)))(\lambda u.u)) \\
 & \\
\xrightarrow{no} & ((\lambda y.((\lambda w.w)(\lambda z.z)))(\lambda u.u))^{\star} \\
\Rightarrow & ((\lambda y.((\lambda w.w)(\lambda z.z)))^{\star}(\lambda u.u)) \\
 & \\
\xrightarrow{no} & ((\lambda w.w)(\lambda z.z))^{\star} \\
\Rightarrow & ((\lambda w.w)^{\star}(\lambda z.z)) \\
 & \\
\xrightarrow{no} & (\lambda z.z)
\end{array}
$$

Normalordnungsreduktion: Eigenschaften (1)

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Die Normalordnungsreduktion ist deterministisch:

Für jeden Ausdruck s gibt es höchstens ein t
(modulo α-Gleichheit),
so dass $s \xrightarrow{no} t$.

Ausdrücke, für die keine Reduktion möglich ist:

- Reduktion stößt auf freie Variable: z.B. $(x\ (\lambda y.y))$
  (nicht bei geschlossenen Ausdrücken)
- Ausdruck ist eine WHNF (weak head normal form), d.h. im Lambda-Kalkül eine Abstraktion.
- Genauer: Ausdruck ist eine FWHNF:
  FWHNF = Abstraktion
  (functional weak head normal form)

Normalordnungsreduktion: Eigenschaften (2)

Weitere Notationen:

$\xrightarrow{no,+}$ = transitive Hülle von $\xrightarrow{no}$
(mehrere $\xrightarrow{no}$ nacheinander)

$\xrightarrow{no,*}$ = reflexiv-transitive Hülle von $\xrightarrow{no}$
(keines oder mehrere $\xrightarrow{no}$ nacheinander)

Definition

Ein Ausdruck s konvergiert ($s{\Downarrow}$ bzw. $s{\Downarrow}v$) gdw.
$\exists v$: v ist FWHNF und $s \xrightarrow{no,*} v$.
Andernfalls divergiert s, Notation $s{\Uparrow}$

Die Begriffe "konvergieren" (zu einem Wert) und "divergieren" (terminiert nicht) sind aus der wiss. Literatur.

Anmerkungen

- (pures) Haskell verwendet den call-by-name Lambda-Kalkül als semantische Grundlage (man braucht noch Erweiterungen . . .)
- Implementierungen verwenden die call-by-need Variante: Vermeidung von Doppelauswertungen (kommt später)
- Call-by-name und call-by-need sind äquivalent (exakte Def kommt noch).
- Call-by-name (und auch call-by-need) sind optimal bzgl. Konvergenz:

Aussage (Standardisierung)

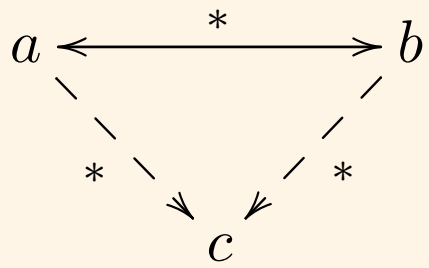
Sei s ein Lambda-Ausdruck. Wenn s mit β-Reduktionen, nacheinander ausgeführt in eine Abstraktion v überführt werden kann, dann gilt $s{\Downarrow}$.

Church-Rosser Theorem

Für den Lambda-Kalkül gilt (unter der Gleichheit $=_\alpha$)

Satz (Konfluenz)

Church-Rosser Eigenschaft:
Wenn $a \overset{*}{\leftrightarrow} b$, dann existiert c, so dass $a \overset{*}{\rightarrow} c$ und $b \overset{*}{\rightarrow} c$



Hierbei bedeutet:
$\overset{*}{\rightarrow}$ Folge von β-Reduktionen, und
$\overset{*}{\leftrightarrow}$ Folge von β-Reduktionen (vorwärts und rückwärts)

Anwendungsordnung (CBV) im Lambda-Kalkül (nicht in Haskell)

Diese ("strikte") Reduktionsstrategie wird verwendet
in Programmiersprachen die strikt auswerten (nicht in Haskell!).

z.B. Wenn `x = 3, y = 2`:
`3 + (x*x - y*y)` $\longrightarrow$ `3 + (9 - 2*2)` $\longrightarrow$ `3 + (9 - 4)`
$\longrightarrow$ `3 + 5` $\longrightarrow$ `8`

D.h. erst das Argument, dann die Funktionsanwendung.

- Sprechweisen: Anwendungsordnung, **call-by-value (CBV)**, **strikt**
- Grobe Umschreibung:
  Argumentauswertung vor Definitionseinsetzung

Anwendungsordnung (CBV)im Lambda-Kalkül

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Call-by-value Beta-Reduktion

$$(\beta_{cbv}) \qquad (\lambda x.s)\ v \to s[v/x],$$

nur wenn v Abstraktion oder Variable

Definition

CBV-Reduktionskontexte $\mathbf{E}$:

$$\mathbf{E} ::= [\cdot] \mid (\mathbf{E}\ \mathbf{Expr}) \mid ((\lambda V.\mathbf{Expr})\ \mathbf{E})$$

Wenn $s \xrightarrow{\beta_{cbv}} t$,

dann ist $E[s] \xrightarrow{ao} E[t]$ eine Anwendungsordnungsreduktion

Anwendungsordnung im Lambda-Kalkül, Bspl.

Call-by-value Beta-Reduktion: Beispiel

$(\lambda x, y, z.x + y + z)\ 1\ ((\lambda x.x)\ 2)\ ((\lambda x.\lambda y.x)\ 3\ 4)$

(Zahlen und Addition mit $+$ nur als Illustration;
diese sind im Lambda-Kalkül nicht vorhanden
bzw. werden speziell kodiert)

Anwendungsordnung im Lambda-Kalkül, Bspl.

Call-by-value Beta-Reduktion: Beispiel

$$(\lambda x, y, z.x + y + z)\ 1\ ((\lambda x.x)\ 2)\ ((\lambda x.\lambda y.x)\ 3\ 4)$$

$$\xrightarrow{ao}$$

$$(\lambda x, y, z.x + y + z)\ 1\ 2\ ((\lambda x.\lambda y.x)\ 3\ 4)$$

(Zahlen und Addition mit $+$ nur als Illustration;
diese sind im Lambda-Kalkül nicht vorhanden
bzw. werden speziell kodiert)

Anwendungsordnung im Lambda-Kalkül, Bspl.

Call-by-value Beta-Reduktion: Beispiel

$$(\lambda x, y, z.x + y + z)\ 1\ ((\lambda x.x)\ 2)\ ((\lambda x.\lambda y.x)\ 3\ 4)$$

$$\xrightarrow{ao}$$

$$(\lambda x, y, z.x + y + z)\ 1\ 2\ ((\lambda x.\lambda y.x)\ 3\ 4)$$

$$\xrightarrow{ao}$$

$$(\lambda x, y, z.x + y + z)\ 1\ 2\ ((\lambda y.3)\ 4)$$

(Zahlen und Addition mit $+$ nur als Illustration;
diese sind im Lambda-Kalkül nicht vorhanden
bzw. werden speziell kodiert)

Anwendungsordnung im Lambda-Kalkül, Bspl.

Call-by-value Beta-Reduktion: Beispiel

$$(\lambda x, y, z.x + y + z)\ 1\ ((\lambda x.x)\ 2)\ ((\lambda x.\lambda y.x)\ 3\ 4)$$

$$\xrightarrow{ao}$$

$$(\lambda x, y, z.x + y + z)\ 1\ 2\ ((\lambda x.\lambda y.x)\ 3\ 4)$$

$$\xrightarrow{ao}$$

$$(\lambda x, y, z.x + y + z)\ 1\ 2\ ((\lambda y.3)\ 4)$$

$$\xrightarrow{ao}$$

$$(\lambda x, y, z.x + y + z)\ 1\ 2\ 3$$

(Zahlen und Addition mit $+$ nur als Illustration;
diese sind im Lambda-Kalkül nicht vorhanden
bzw. werden speziell kodiert)

Anwendungsordnung im Lambda-Kalkül, Bspl.

Call-by-value Beta-Reduktion: Beispiel

$$(\lambda x, y, z.x + y + z)\ 1\ ((\lambda x.x)\ 2)\ ((\lambda x.\lambda y.x)\ 3\ 4)$$

$$\xrightarrow{ao}$$

$$(\lambda x, y, z.x + y + z)\ 1\ 2\ ((\lambda x.\lambda y.x)\ 3\ 4)$$

$$\xrightarrow{ao}$$

$$(\lambda x, y, z.x + y + z)\ 1\ 2\ ((\lambda y.3)\ 4)$$

$$\xrightarrow{ao}$$

$$(\lambda x, y, z.x + y + z)\ 1\ 2\ 3$$

$$\xrightarrow{ao}$$

$$(\lambda y, z.1 + y + z)\ 2\ 3$$ usw.

(Zahlen und Addition mit $+$ nur als Illustration;
diese sind im Lambda-Kalkül nicht vorhanden
bzw. werden speziell kodiert)

CBV-Redexsuche mit Markierungsalgorithmus

- Starte mit $s^\star$ (s keine Abstraktion oder Variable)
- Wende die folgenden Regeln an solange es geht:
  - $(s_1\ s_2)^\star \Rightarrow (s_1^\star\ s_2)$
  - $(v^\star\ s_1) \Rightarrow (v\ {s_1}^\star)$
    falls v eine Abstraktion und

CBV-Redexsuche mit Markierungsalgorithmus

- Starte mit $s^\star$ (s keine Abstraktion oder Variable)
- Wende die folgenden Regeln an solange es geht:
  - $(s_1\ s_2)^\star \Rightarrow (s_1^\star\ s_2)$
  - $(v^\star\ s_1) \Rightarrow (v\ {s_1}^\star)$
    falls v eine Abstraktion und
- Beispiel: $((((\lambda x.x)\ (((\lambda y.y)\ v)\ (\lambda z.z)))\ u)\ (\lambda w.w))^\star$

CBV-Redexsuche mit Markierungsalgorithmus

- Starte mit $s^\star$ (s keine Abstraktion oder Variable)
- Wende die folgenden Regeln an solange es geht:
  - $(s_1\ s_2)^\star \Rightarrow (s_1^\star\ s_2)$
  - $(v^\star\ s_1) \Rightarrow (v\ s_1{}^\star)$
    falls v eine Abstraktion und
- Beispiel: $((((\lambda x.x)\ (((\lambda y.y)\ v)\ (\lambda z.z)))\ u)^\star\ (\lambda w.w))$

CBV-Redexsuche mit Markierungsalgorithmus

- Starte mit $s^\star$ (s keine Abstraktion oder Variable)
- Wende die folgenden Regeln an solange es geht:
  - $(s_1\ s_2)^\star \Rightarrow (s_1^\star\ s_2)$
  - $(v^\star\ s_1) \Rightarrow (v\ {s_1}^\star)$
    falls v eine Abstraktion und
- Beispiel: $((((\lambda x.x)\ (((\lambda y.y)\ v)\ (\lambda z.z)))^\star\ u)\ (\lambda w.w))$

CBV-Redexsuche mit Markierungsalgorithmus

- Starte mit $s^\star$ (s keine Abstraktion oder Variable)
- Wende die folgenden Regeln an solange es geht:
  - $(s_1\ s_2)^\star \Rightarrow (s_1^\star\ s_2)$
  - $(v^\star\ s_1) \Rightarrow (v\ s_1{}^\star)$
    falls v eine Abstraktion und
- Beispiel: $((((\lambda x.x)^\star\ (((\lambda y.y)\ v)\ (\lambda z.z)))\ u)\ (\lambda w.w))$

CBV-Redexsuche mit Markierungsalgorithmus

- Starte mit $s^\star$ (s keine Abstraktion oder Variable)
- Wende die folgenden Regeln an solange es geht:
  - $(s_1\ s_2)^\star \Rightarrow (s_1^\star\ s_2)$
  - $(v^\star\ s_1) \Rightarrow (v\ {s_1}^\star)$
    falls v eine Abstraktion und
- Beispiel: $((((\lambda x.x)\ (((\lambda y.y)\ v)\ (\lambda z.z))^\star)\ u)\ (\lambda w.w))$

CBV-Redexsuche mit Markierungsalgorithmus

- Starte mit $s^\star$ (s keine Abstraktion oder Variable)
- Wende die folgenden Regeln an solange es geht:
  - $(s_1\ s_2)^\star \Rightarrow (s_1^\star\ s_2)$
  - $(v^\star\ s_1) \Rightarrow (v\ {s_1}^\star)$
    falls v eine Abstraktion und
- Beispiel: $((((\lambda x.x)\ (((\lambda y.y)\ v)^\star\ (\lambda z.z)))\ u)\ (\lambda w.w))$

CBV-Redexsuche mit Markierungsalgorithmus

- Starte mit $s^\star$ (s keine Abstraktion oder Variable)
- Wende die folgenden Regeln an solange es geht:
  - $(s_1\ s_2)^\star \Rightarrow (s_1^\star\ s_2)$
  - $(v^\star\ s_1) \Rightarrow (v\ s_1{}^\star)$
    falls v eine Abstraktion und
- Beispiel: $((((\lambda x.x)\ (((\lambda y.y)^\star\ v)\ (\lambda z.z)))\ u)\ (\lambda w.w))$

CBV-Redexsuche mit Markierungsalgorithmus

- Starte mit $s^\star$ (s keine Abstraktion oder Variable)
- Wende die folgenden Regeln an solange es geht:
  - $(s_1\ s_2)^\star \Rightarrow (s_1^\star\ s_2)$
  - $(v^\star\ s_1) \Rightarrow (v\ s_1{}^\star)$
    falls v eine Abstraktion und
- Beispiel: $((((\lambda x.x)\ (((\lambda y.y)^\star\ v)\ (\lambda z.z)))\ u)\ (\lambda w.w))$

Falls danach gilt: $C[(\lambda x.s)^\star\ v]$ dann

$$C[(\lambda x.s)^\star v] \xrightarrow{ao} C[s[v/x]]$$

Also $C[((\lambda y.y)^\star\ v)] \xrightarrow{ao} C[v]$

Beispiel

$$
\begin{array}{ll}
 & (((\lambda x.\lambda y.x)((\lambda w.w)(\lambda z.z)))(\lambda u.u))^{\star} \\
\Rightarrow & (((\lambda x.\lambda y.x)((\lambda w.w)(\lambda z.z)))^{\star}(\lambda u.u)) \\
\Rightarrow & (((\lambda x.\lambda y.x)^{\star}((\lambda w.w)(\lambda z.z)))(\lambda u.u)) \\
\Rightarrow & (((\lambda x.\lambda y.x)((\lambda w.w)(\lambda z.z))^{\star})(\lambda u.u)) \\
\Rightarrow & (((\lambda x.\lambda y.x)((\lambda w.w)^{\star}(\lambda z.z)))(\lambda u.u)) \\
 & \\
\xrightarrow{ao} & (((\lambda x.\lambda y.x)(\lambda z.z))(\lambda u.u))^{\star} \\
\Rightarrow & (((\lambda x.\lambda y.x)(\lambda z.z))^{\star}(\lambda u.u)) \\
\Rightarrow & (((\lambda x.\lambda y.x)^{\star}(\lambda z.z))(\lambda u.u)) \\
 & \\
\xrightarrow{ao} & ((\lambda y.\lambda z.z)(\lambda u.u))^{\star} \\
\Rightarrow & ((\lambda y.\lambda z.z)^{\star}(\lambda u.u)) \\
 & \\
\xrightarrow{ao} & (\lambda z.z)
\end{array}
$$

Eigenschaften

- Auch die call-by-value Reduktion ist deterministisch.

Definition

Ein Ausdruck s call-by-value konvergiert ($s \Downarrow_{ao}$), gdw.
$\exists$ FWHNF $v : s \xrightarrow{ao,*} v$.
Ansonsten (call-by-value) divergiert s, Notation: $s \Uparrow_{ao}$.

Eigenschaften

- Auch die call-by-value Reduktion ist deterministisch.

Definition

Ein Ausdruck s call-by-value konvergiert ($s \Downarrow_{ao}$), gdw.
$\exists$ FWHNF $v : s \xrightarrow{ao,*} v$.
Ansonsten (call-by-value) divergiert s, Notation: $s \Uparrow_{ao}$.

- Es gilt: $s \Downarrow_{ao} \implies s \Downarrow$.
- Die Umkehrung gilt nicht!
  $((\lambda y.(\lambda x.x))\ \Omega) \Uparrow_{ao}$ Wobei: $\Omega = (\lambda x.x\ x)\ (\lambda x.x\ x)$
  aber
  $(\lambda y.(\lambda x.x))\ \Omega \Downarrow_{no}$ da: $((\lambda y.(\lambda x.x))\ \Omega) \rightarrow_{no} (\lambda x.x)$

Eigenschaften

Vorteile der Anwendungsordnung (call-by-value):

- Tlw. besseres Platzverhalten (als call-by-name)
- Auswertungsreihenfolge liegt fest:
  (im Syntaxbaum von Funktionen f); bzw. ist innerhalb jeder Funktionsdefinition vorhersagbar.
  Bei AO: $f\ s_1\ s_2\ s_3$: immer zuerst s_1, dann s_2, dann s_3, dann $(f\ s_1\ s_2\ s_3)$
  Bei NO: zum Beispiel: zuerst s_1, dann evtl. abhängig vom Wert von s_1:
  zuerst s_2, dann s_3, oder andersrum,
  oder evtl. weder s_2 noch s_3.
  Dazwischen auch Auswertung von Anteilen von $(f\ s_1\ s_2\ s_3)$.
- Wegen der vorhersagbaren Auswertungsreihenfolge (in Funktionsdefinitionen) unter AO:
  Seiteneffekte können unter AO compiliert werden; und sind nicht abhängig von Bedingungen.

Haskell

Ab jetzt: Haskell:

Konstrukte, Eigenschaften und Beispiele

In Haskell: seq

In Haskell: (Lokale) strikte Auswertung kann mit seq erzwungen werden.

$$\begin{array}{ll} \texttt{seq}\ a\ b = b & \text{falls } a \Downarrow \\ (\texttt{seq}\ a\ b) \Uparrow & \text{falls } a \Uparrow \end{array}$$

- in Anwendungsordung ist seq kodierbar, aber unnötig
- in Normalordnung:

  seq kann nicht im Lambda-Kalkül kodiert werden!

  seq kann auch nicht in erweiterten Kernsprachen kodiert werden!

  seq muss also **explizit** in Haskell eingebaut werden.

Beispiel zu seq in Haskell

```
fak 0 = 1
fak x = x * fak(x-1)
```

Auswertung von `fak n`:

```
     fak n
 →   n * fak (n-1)
 →   n * ((n-1) * fak (n-2))
 →   ...
 →   n * ((n-1) * ((n-2) * .... * (2 * 1)))
 →   n * ((n-1) * ((n-2) * .... * 2))
 →   ...
```

Problem: Linearer Platzbedarf

Beispiel zu seq (2)

Version mit seq:

```
fak' x     = fak'' x 1

fak'' 0 y = y
fak'' x y = let m = x*y in seq m (fak'' (x-1) m)
```

Auswertung in etwa:

```
      fak' 5
→     fak'' 5 1
→     let m=5*1 in seq m (fak'' (5-1) m)
→     let m=5 in seq m (fak'' (5-1) m)
→     (fak'' (5-1) 5)
→     (fak'' 4 5)
→     let m = 4*5 in seq m (fak'' (4-1) m)
→     let m = 20 in seq m (fak'' (4-1) m)
→     (fak'' (4-1) 20)
→      ...
```

Jetzt: konstanter Platzbedarf, da Zwischenprodukt `m` berechnet wird (erzwungen durch `seq`) und sharing mittels `let`.

Beispiele zu Normal- und Anwendungsordnung

Im Lambdakalkül.

Nicht in Haskell !!

Nur im Lambdakalkül programmierbar!!

- $\Omega := (\lambda x.x\ x)\ (\lambda x.x\ x)$.
- $\Omega \xrightarrow{no} \Omega$. Daraus folgt: $\Omega \Uparrow$
- $\Omega \xrightarrow{ao} \Omega$. Daraus folgt: $\Omega \Uparrow_{ao}$.
- $t := ((\lambda x.(\lambda y.y))\ \Omega)$.
- $t \xrightarrow{no} \lambda y.y$, d.h. $t \Downarrow$.
- $t \xrightarrow{ao} t$, also $t \Uparrow_{ao}$ denn die Anwendungsordnung muss zunächst das Argument Ω auswerten.

Fazit: Auswertungen mit Normalordnung und Anwendungsordnung haben verschiedenes Terminierungsverhalten

Hinweise zur **verzögerten Auswertung**

- Sprechweisen: Verzögerte Auswertung, **call-by-need**, **nicht-strikt**, **lazy**, (Reduktion mit Sharing)
- Optimierung der Normalordnungsreduktion

Verzögerte Auswertung kann durch Erweiterung des Lambda-Kalküls mit `let` modelliert werden:
Call-by-need Lambda-Kalkül mit `let`– Syntax:

$$\mathbf{Expr} ::= V \mid \lambda V.\mathbf{Expr} \mid (\mathbf{Expr}\ \mathbf{Expr}) \mid \texttt{let}\ V = \mathbf{Expr}\ \texttt{in}\ \mathbf{Expr}$$

- Zur Modellierung des sharing (von call-by-need) reicht die einfachere Variante eines nicht-rekursiven `let`:
  $\texttt{let}\ x = s\ \texttt{in}\ t$ wobei $x \notin FV(s)$ (siehe Skript)
- Aber: Haskell verwendet rekursives `let`!
  (deswegen komplizierter zu beschreiben und zu erklären)

Einschub: Programmieren mit `let` in Haskell

- `let` in Haskell ist viel allgemeiner als das einfache nicht-rekursive `let`.
- Haskells `let` für lokale Funktionsdefinitionen:

$$\begin{array}{lll} \texttt{let} & f_1\ x_{1,1}\ \ldots\ x_{1,n_1} & = e_1 \\ & f_2\ x_{2,1}\ \ldots\ x_{2,n_2} & = e_2 \\ & \ldots & \\ & f_m\ x_{m,1}\ \ldots\ x_{m,n_m} & = e_m \\ \texttt{in} \ldots & & \end{array}$$

Definiert die Funktionen $f_1, \ldots, f_m$ (auch verschränkt rekursiv)

Beispiel:

```
f x y =  let quadrat z = z*z
         in quadrat x + quadrat y
```

Rekursives `let` in Haskell

Verschränkt-rekursives `let` erlaubt:

```
quadratfakultaet x =
   let quadrat z = z*z
       fakq    0 = 1
       fakq    x = (quadrat x)*fakq (x-1)
   in fakq x
```

Sharing von Ausdrücken mittels `let`:

```
verdopplefak x =
 let fak 0  = 1
     fak x  = x*fak (x-1)
     fakx   = fak x
 in fakx + fakx
```

```
verdopplefakLangsam x =
  let fak 0  = 1
      fak x  = x*fak (x-1)
  in fak x + fak x
```

`verdopplefak 100` berechnet nur einmal `fak 100`, im Gegensatz zu `verdopplefakLangsam`.

Pattern-Matching mit `let`

Links in einer `let`-Bindung in Haskell darf auch ein Pattern stehen.

Beispiel: $\sum_{i=1}^{n} i$ und $\prod_{i=1}^{n} i$ in einer rekursiven Funktion:

```
sumprod 1 = (1,1)
sumprod n =  let (s',p') = sumprod (n-1)
             in (s'+n,p'*n)
```

Das Paar `(s',p')` aus dem rekursiven Aufruf versucht Pattern Matching auf Ausdrücke, zu denen `sumprod (n-1)` auswertet.

Memoization

Beispiel: Fibonacci-Zahl (einfach aber ineffizient programmiert).
1,1,2,3,5,8,13,21,34, ...

```
fib 0 = 0
fib 1 = 1
fib i = fib (i-1) + fib (i-2)
```

sehr schlechte Laufzeit!

n	gemessene Zeit im ghci für `fib` n
30	9.75sec
31	15.71sec
32	25.30sec
33	41.47sec
34	66.82sec
35	108.16sec

Memoization (2)

Besser:

```
-- Fibonacci mit let und verbessertem sharing
fibM i =
   let fibs = [0,1] ++ [fibs!!(i-1) + fibs!!(i-2) | i <- [2..]]
   in fibs!!i -- i-tes Element der Liste fibs
```

n	gemessene Zeit im ghci für `fibM` n
1000	0.05sec
10000	1.56sec
20000	7.38sec
30000	23.29sec

Bei mehreren Aufrufen, noch besser:

```
fibM'  =
   let fibs = [0,1] ++ [fibs!!(i-1) + fibs!!(i-2) | i <- [2..]]
   in \i -> fibs!!i -- i-tes Element der Liste fibs
```

where-Ausdrücke (1)

`where`-Ausdrücke sind ähnlich zu `let`.

Z.B.

```
sumprod' 1 = (1,1)
sumprod' n = (s'+n,p'*n)
  where (s',p') = sumprod' (n-1)
```

where-Ausdrücke (2)

Beachte: (`let ... in` e) ist ein Ausdruck, aber e `where ...` nicht

`where` kann man um Guards herum schreiben (`let` nicht):

```
f x
  | x == 0     = a
  | x == 1     = a*a
  | otherwise = a*f (x-1)
  where a = 10
```

Dafür geht

```
f x = \y -> mul
    where mul = x * y
```

nicht! (da `y` nicht bekannt in der `where`-Deklaration)

Gleichheit von Ausdrücken (semantisch)

Praktischer Nutzen einer Äquivalenz auf Ausdrücken:
Optimierung durch Abändern von Teilausdrücken $P[s] \to P[s']$ ist eine erlaubte Transformation, wenn s und s' "äquivalent" sind.

- Wann sind zwei Ausdrücke **gleich(wertig)**?
- D.h. insbesondere: Wann darf ein Compiler einen Ausdruck durch einen anderen ersetzen?
- Das hängt ab von der Programmiersprache und Auswertung.

Kalküle bisher in der Vorlesung:

- Call-by-Name Lambda-Kalkül: Ausdrücke, $\xrightarrow{no}$, $\Downarrow$
- Call-by-Value Lambda-Kalkül: Ausdrücke, $\xrightarrow{ao}$, $\Downarrow_{ao}$
- (Call-by-Need Lambda-Kalkül . . .)

D.h. Syntax + Operationale Semantik.

Gleichheit (2)

- Leibnizsches Prinzip: Zwei Dinge sind gleich, wenn sie die gleichen Eigenschaften haben, bzgl. (fast) aller Eigenschaften.
- Für Programm-Kalküle: Zwei Ausdrücke s, t sind gleich, wenn man sie nicht unterscheiden kann, (bzgl Ergebnis, bzw. Terminierung, ...), egal in welchem Kontext man sie benutzt.
- Formaler: s,t sind gleich, wenn für alle Kontexte C gilt:
  $C[s]$ und $C[t]$ verhalten sich gleich.
- **Verhalten** muss noch definiert werden. Für deterministische Sprachen reicht die Beobachtung der Terminierung bzw. Konvergenz
- Erweitertes Kriterium: gleicher Effekt, aber effizienter: Optimierung.
  Aber: es ist i.a schwer nachzuweisen, das eine Änderung niemals verschlechtert.

Gleichheit (2)

- Leibnizsches Prinzip: Zwei Dinge sind gleich, wenn sie die gleichen Eigenschaften haben, bzgl. (fast) aller Eigenschaften.
- Für Programm-Kalküle: Zwei Ausdrücke s, t sind gleich, wenn man sie nicht unterscheiden kann, (bzgl Ergebnis, bzw. Terminierung, ...), egal in welchem Kontext man sie benutzt.
- Formaler: s,t sind gleich, wenn für alle Kontexte C gilt:
  $C[s]$ und $C[t]$ verhalten sich gleich.
- **Verhalten** muss noch definiert werden. Für deterministische Sprachen reicht die Beobachtung der Terminierung bzw. Konvergenz
- Erweitertes Kriterium: gleicher Effekt, aber effizienter: Optimierung.
  Aber: es ist i.a schwer nachzuweisen, das eine Änderung niemals verschlechtert.

Gleichheit (2)

- Leibnizsches Prinzip: Zwei Dinge sind gleich, wenn sie die gleichen Eigenschaften haben, bzgl. (fast) aller Eigenschaften.
- Für Programm-Kalküle: Zwei Ausdrücke s, t sind gleich, wenn man sie nicht unterscheiden kann, (bzgl Ergebnis, bzw. Terminierung, ...), egal in welchem Kontext man sie benutzt.
- Formaler: s,t sind gleich, wenn für alle Kontexte C gilt:
  $C[s]$ und $C[t]$ verhalten sich gleich.
- **Verhalten** muss noch definiert werden. Für deterministische Sprachen reicht die Beobachtung der Terminierung bzw. Konvergenz
- Erweitertes Kriterium: gleicher Effekt, aber effizienter: Optimierung.
  Aber: es ist i.a schwer nachzuweisen, das eine Änderung niemals verschlechtert.

Gleichheit (2)

- Leibnizsches Prinzip: Zwei Dinge sind gleich, wenn sie die gleichen Eigenschaften haben, bzgl. (fast) aller Eigenschaften.
- Für Programm-Kalküle: Zwei Ausdrücke s, t sind gleich, wenn man sie nicht unterscheiden kann, (bzgl Ergebnis, bzw. Terminierung, ...), egal in welchem Kontext man sie benutzt.
- Formaler: s,t sind gleich, wenn für alle Kontexte C gilt:
  $C[s]$ und $C[t]$ verhalten sich gleich.
- **Verhalten** muss noch definiert werden. Für deterministische Sprachen reicht die Beobachtung der Terminierung bzw. Konvergenz
- Erweitertes Kriterium: gleicher Effekt, aber effizienter: Optimierung.
  Aber: es ist i.a schwer nachzuweisen, das eine Änderung niemals verschlechtert.

Gleichheit wozu?

- Wenn s als Ausdruck in einem Programm vorkommt und $s \sim t$, dann kann man s durch t ersetzen.
- Wenn man zusätzlich weiß,
  dass t schneller ist als s (bzw. $P[t]$ als $P[s]$),
  dann ist t eine Optimierung von s (bzw. $P[t]$ von $P[s]$)
- Korrektheit und Effizienzverbesserung (einer solchen Transformation) hängen davon ab, welche operationale Semantik die Programmiersprache hat.
- Man kann verschiedene solche Transformationen nacheinander verwenden.

Gleichheit von Programmen

In deterministischen (funktionalen) Programmiersprachen:

- P(Eingabe) ist äquivalent zum berechneten Ergebnis.

In nicht-deterministischen (funktionalen) Programmiersprachen:

- P(Eingabe) ist äquivalent
  zur Veroderung aller Ergebnisse
  $\oplus$(Resultate)

Vorsicht: muss man genauer definieren.

Gleichheit (3)

Kontextuelle Approximation und Gleichheit

Call-by-Name Lambda-Kalkül:

- $s \leq_c t$ gdw. $\forall C : C[s] \Downarrow \Longrightarrow C[t] \Downarrow$

Gleichheit (3)

Kontextuelle Approximation und Gleichheit

Call-by-Name Lambda-Kalkül:

- $s \leq_c t$ gdw. $\forall C : C[s] \Downarrow \Longrightarrow C[t] \Downarrow$
- $s \sim_c t$ gdw. $s \leq_c t$ und $t \leq_c s$

Gleichheit (3)

Kontextuelle Approximation und Gleichheit

Call-by-Name Lambda-Kalkül:

- $s \leq_c t$ gdw. $\forall C : C[s] \Downarrow \Longrightarrow C[t] \Downarrow$
- $s \sim_c t$ gdw. $s \leq_c t$ und $t \leq_c s$

Call-by-Value Lambda-Kalkül:

- $s \leq_{c,ao} t$ gdw. $\forall C$: Wenn $C[s], C[t]$ geschlossen sind und $C[s] \Downarrow_{ao}$, dann auch $C[t] \Downarrow_{ao}$.
- $s \sim_{c,ao} t$ gdw. $s \leq_{c,ao} t$ und $t \leq_{c,ao} s$

Gleichheit (4)

- $\sim_c$ und $\sim_{ao}$ sind Kongruenzen
- Kongruenz = Äquivalenzrelation + kompatibel mit Kontexten, d.h. $s \sim t \implies C[s] \sim C[t]$.
- Korrektheit einer konkreten Transformation zu beweisen ist i.a. schwer. (Das Widerlegen ist oft einfacher)

Gleichheit (4)

- $\sim_c$ und $\sim_{ao}$ sind Kongruenzen
- Kongruenz = Äquivalenzrelation + kompatibel mit Kontexten, d.h. $s \sim t \implies C[s] \sim C[t]$.
- Korrektheit einer konkreten Transformation zu beweisen ist i.a. schwer. (Das Widerlegen ist oft einfacher)

Anmerkung zu „$C[s], C[t]$ geschlossen“ bei $\sim$

- $\sim_c$ ändert sich nicht beim Übergang auf $C[s], C[t]$ geschlossen, aber $\sim_{c,ao}$.
- $\sim_{c,ao}$ wird auch in erweiterten Kalkülen so definiert, und:
- $\sim_{c,ao}$ hat unter der closed-Bedingung mehr sinnvolle Gleichheiten als ohne diese Bedingung (in den erweiterten Kalkülen)

Gleichheit (5)

Beispiele für Gleichheiten, Ungleichheiten, Implikation von Relationen:

- $(\lambda x, y.x)(\lambda z.z) \sim_c \lambda y.(\lambda z.z)$,
  Allgemeiner:
- $(\beta) \subseteq \sim_c$ (lambda-Kalkül mit Normalordnung)
- $\sim_c \not\subseteq \sim_{c,ao}$ und $\sim_{c,ao} \not\subseteq \sim_c$
- $(\beta_{cbv}) \subseteq \sim_{c,ao}$ aber $(\beta) \not\subseteq \sim_{c,ao}$

Gleichheit (6)

Praktische Bedeutung der Gleichheit:

- Compiler darf Gleiches durch Gleiches ersetzen!
  (Möglichst durch besseren/schnelleren Code ersetzen)
- Kann zur Optimierung verwendet werden.
- Man braucht noch Wissen darüber,
  bzw. welche Variante des neuen Codes unter verschiedenen Bedingungen effizienter ist.