Semantik funktionaler Programmiersprachen: Funktionale Kernsprachen

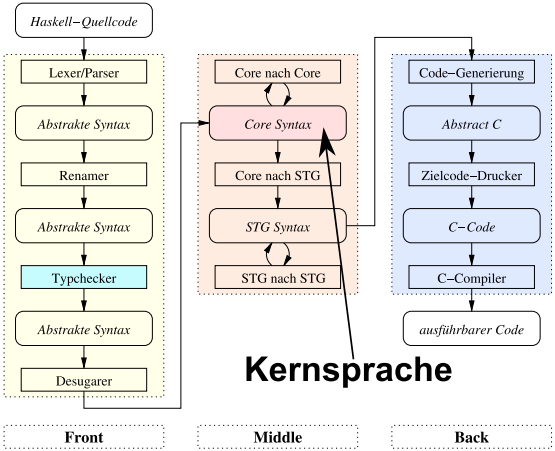
Prof. Dr. David Sabel

LFE Theoretische Informatik

Letzte Änderung der Folien: 19. November 2021 ♦ Die Folien basieren zum Teil auf Material von Dr. Steffen Jost, dem an dieser Stelle für die Verwendungserlaubnis herzlich gedankt sei.

Einleitung

Compilerphasen des GHC (schematisch)



Überblick

- Als Ziel: Definition einer Kernsprache, die zu Haskell passt
- Vorgehen:
  - Wir betrachten zunächst den Lambda-Kalkül
    - Er ist „Kernsprache“ fast aller (funktionalen) Programmiersprachen
    - Allerdings oft zu minimalistisch
  - Betrachte 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 quasi einen Interpreter
- Verschiedene Formalismen:
  - Zustandsübergangssysteme
  - Abstrakte Maschinen
  - Ersetzungssysteme
- Unterscheidung in small-step und big-step Semantiken
- Wir verwenden operationale (small-step) Semantiken

Der Lambda-Kalkül

Syntax, Call-by-Name-Reduktion, Call-by-Value-Reduktion,
Call-by-Need-Reduktion, Gleichheit

Der Lambda-Kalkül

- Von Alonzo Church und Stephen Kleene in den 1930er Jahren eingeführt
- Wir betrachten den ungetypten Lambda-Kalkül.
- Idee: Darstellungen der Berechnung von Funktionen
- Der ungetypte Lambda-Kalkül ist Turing-mächtig.
- Viele Ausdrücke des Lambda-Kalküls können in Haskell eingegeben werden, aber nur wenn sie Haskell-typisierbar sind

Syntax des Lambda-Kalküls

$$
\begin{array}{rcll}
\mathbf{Expr} & ::= & V & \text{Variable (unendliche Menge)} \\
& | & \lambda V.\mathbf{Expr} & \text{Abstraktion} \\
& | & (\mathbf{Expr}\ \mathbf{Expr}) & \text{Anwendung (Applikation)}
\end{array}
$$

- Abstraktionen sind anonyme Funktionen

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

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

  s, t beliebige Ausdrücke $\implies$ 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 $\lambda x.\lambda y.\lambda z.s$

Syntax des Lambda-Kalküls (3)

Prominente Ausdrücke

I	$:=$	$\lambda x.x$	Identität
K	$:=$	$\lambda x.\lambda y.x$	Projektion auf Argument 1
$K2$	$:=$	$\lambda x.\lambda y.y$	Projektion auf Argument 2
Ω	$:=$	$(\lambda x.(x\ x))\ (\lambda x.(x\ x))$	terminiert nicht
Y	$:=$	$\lambda f.(\lambda x.(f\ (x\ x)))\ (\lambda x.(f\ (x\ x)))$	Fixpunkt-Kombinator
S	$:=$	$\lambda x.\lambda y.\lambda z.(x\ z)\ (y\ z)$	S-Kombinator

Bemerkung: Mit den Kombinatoren S,K,I alleine kann man eine Turing-mächtige Sprache erhalten (siehe SKI combinator calculus)

Freie und gebundene Vorkommen von 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
- anderenfalls 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

Freie und gebundene Variablen

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

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

Freie und gebundene Variablen

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

Anderenfalls: t ist ein offener Ausdruck

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

Substitution

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

Formale Definition

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

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

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

Kontexte

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

Als Grammatik

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

- Sei C ein Kontext und s ein Ausdruck s:

  $C[s]$ = 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 gleich.
- z.B. $\lambda x.x =_\alpha \lambda y.y$
- Distinct Variable Convention: Alle gebundenen Variablen sind verschieden und gebundene Variablen sind verschieden von freien.
- α-Umbenennungen ermöglichen, dass die DVC stets erfüllt werden kann.

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.

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

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

Operationale Semantik - Beta-Reduktion

Beta-Reduktion

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

Wenn $r_1 \xrightarrow{\beta} r_2$, dann sagt man: r_1 reduziert unmittelbar zu r_2.

Operationale Semantik - Beta-Reduktion

Beta-Reduktion

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

Wenn $r_1 \xrightarrow{\beta} r_2$, dann sagt man: r_1 reduziert unmittelbar zu r_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:

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

Operationale Semantik

- Für die Festlegung der operationalen Semantik muss man noch definieren, wo in einem Ausdruck die β-Reduktion angewendet wird
- Betrachte $((\lambda x.x\ x)((\lambda y.y)(\lambda z.z)))$.

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

Call-by-Name-Reduktion

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

Call-by-Name-Reduktion

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

Definition

- Reduktionskontexte R erzeugt durch $\mathbf{RCtxt} ::= [\cdot] \mid (\mathbf{RCtxt}\ \mathbf{Expr})$
- $\xrightarrow{name}$: Wenn $r_1 \xrightarrow{\beta} r_2$ und R ein Reduktionskontext, dann ist

$$R[r_1] \xrightarrow{name} R[r_2]$$

ein Call-by-Name-Reduktionsschritt

Call-by-Name-Reduktion

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

Definition

- Reduktionskontexte R erzeugt durch $\mathbf{RCtxt} ::= [\cdot] \mid (\mathbf{RCtxt}\ \mathbf{Expr})$
- $\xrightarrow{name}$: Wenn $r_1 \xrightarrow{\beta} r_2$ und R ein Reduktionskontext, dann ist

$$R[r_1] \xrightarrow{name} R[r_2]$$

ein Call-by-Name-Reduktionsschritt

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: $\mathbf{RCtxt} ::= [\cdot] \mid (\mathbf{RCtxt}\ \mathbf{Expr})$

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

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: $\mathbf{RCtxt} ::= [\cdot] \mid (\mathbf{RCtxt}\ \mathbf{Expr})$

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

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: $((\lambda w.w)\ (\lambda y.y)) \xrightarrow{\beta} (\lambda y.y)$.
  $s = R[((\lambda w.w)\ (\lambda y.y))] \xrightarrow{name} R[\lambda y.y] = ((\lambda y.y)\ ((\lambda z.(\lambda x.x)\ z)\ u))$

Reduktionskontexte: Beispiele

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

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

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: $((\lambda w.w)\ (\lambda y.y)) \xrightarrow{\beta} (\lambda y.y)$.
  $s = R[((\lambda w.w)\ (\lambda y.y))] \xrightarrow{name} R[\lambda y.y] = ((\lambda y.y)\ ((\lambda z.(\lambda x.x)\ z)\ u))$
- $R = ([\cdot]\ (\lambda y.y))\ ((\lambda z.(\lambda x.x)\ z)\ u)$,
  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:

$$((\lambda x.t)\ s_2\ \ldots s_n) \xrightarrow{name} (t[s_2/x]\ \ldots s_n)$$

Beispiel

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

Call-by-Name-Reduktion: Eigenschaften (1)

Die Call-by-Name-Reduktion ist deterministisch:

Für jeden Ausdruck s gibt es höchstens ein t, so dass $s \xrightarrow{name} t$.

Ausdrücke, für die keine Reduktion möglich ist:

- Reduktion stößt auf freie Variable: z.B. $(x\ (\lambda y.y))$
- 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)

Call-by-Name-Reduktion: Eigenschaften (2)

Weitere Notationen:

$\xrightarrow{name,+}$ = transitive Hülle von $\xrightarrow{name}$

$\xrightarrow{name,*}$ = reflexiv-transitive Hülle von $\xrightarrow{name}$

Definition

Ein Ausdruck s konvergiert ($s\downarrow$) gdw. $\exists$FWHNF $v : s \xrightarrow{name,*} v$.
Andernfalls divergiert s, Notation $s\Uparrow$

Anmerkungen

- Haskell verwendet den call-by-name Lambda-Kalkül als semantische Grundlage
- Implementierungen verwenden call-by-need Variante: Vermeidung von Doppelauswertungen (kommt später)
- Call-by-name (und auch call-by-need) sind optimal bzgl. Konvergenz:

Aussage (Standardisierung)

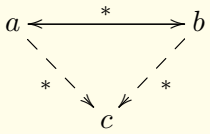
Sei s ein Lambda-Ausdruck. Wenn s mit beliebigen β-Reduktionen (an beliebigen Positionen) in eine Abstraktion v überführt werden kann, dann gilt $s\downarrow$.

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 $a \overset{*}{\to} c$ und $b \overset{*}{\to} c$



Hierbei bedeutet:
$\overset{*}{\to}$ beliebige Folge von β-Reduktionen (in beliebigem Kontext), und
$\overset{*}{\leftrightarrow}$ beliebige Folge von β-Reduktionen (vorwärts und rückwärts) (in beliebigem Kontext)

Implementierung eines Interpreters für den Lambda-Kalkül (1)

Datentyp für Ausdrücke:

```
data LExp v = Var v                    -- x
            | Lambda v (LExp v)        -- \x.s
            | App (LExp v) (LExp v)    -- (s t)
 deriving(Eq,Show)
```

Polymorph über den Namen der Variablen, wir werden im folgenden Strings hierfür verwenden.
Beispiel $s = (\lambda x.x)\ (\lambda y.y)$:

```
s :: LExp String
s =  App (Lambda "x" (Var "x")) (Lambda "y" (Var "y"))
```

Implementierung eines Interpreters für den Lambda-Kalkül (2)

Funktion `rename`: Umbenennung eines Ausdrucks mit frischen Variablennamen

```
rename :: (Eq b) => LExp b -> [b] -> (LExp b, [b])
rename expr vars = rename_it expr [] vars
 where
  rename_it (Var v) renamings vars          =
     case lookup v renamings of
         Nothing -> (Var v,vars)
         Just v'  -> (Var v',vars)
  rename_it (App e1 e2) renamings vars      =
     let (e1',vars') = rename_it e1 renamings vars
         (e2',vars'') = rename_it e2 renamings vars'
     in (App e1' e2', vars'')
  rename_it (Lambda v e) renamings (f:vars) =
     let (e',vars') = rename_it e ((v,f):renamings) vars
     in (Lambda f e',vars')
```

- Eingabe Ausdruck und Liste neuer Namen
- Ausgabe: umbenannter Ausdruck und übrige Namen
- `renamings` enthält die momentan Umbenennung als Liste von Paaren (alter Name, neuer Name)

Implementierung eines Interpreters für den Lambda-Kalkül (3)

Hilfsfunktion `substitute`: führt die Substitution $s[t/x]$ durch
Um die DVC einzuhalten, wird jedes eingesetzte t (mit `rename` frisch umbenannt)

```
-- substitute freshvars expr1 expr2 var f"uhrt die Ersetzung expr1[expr2/var] durch
substitute  :: (Eq b) => [b] -> LExp b -> LExp b -> b -> (LExp b, [b])
substitute freshvars (Var v) expr2 var
 | v == var =  rename (expr2) freshvars
 | otherwise = (Var v,freshvars)
substitute freshvars (App e1 e2) expr2 var =
 let (e1',vars')  = substitute freshvars e1 expr2 var
     (e2',vars'') = substitute vars' e2 expr2 var
 in (App e1' e2', vars'')
substitute freshvars (Lambda v e) expr2 var =
 let (e',vars')   = substitute freshvars e expr2 var
 in (Lambda v e',vars')
```

- DVC muss für die Eingaben gelten (sonst wäre Fall `Lambda v e` aufwändiger).

Implementierung eines Interpreters für den Lambda-Kalkül (4)

Ein Call-by-Name-Reduktionsschritt, falls möglich

```
-- Einfachster Fall: Beta-Reduktion ist auf Top-Level m"oglich:
tryNameBeta (App (Lambda v e) e2) freshvars =
 let (e',vars) = substitute freshvars e e2 v
 in Just (e',vars)

-- Andere Anwendungen: gehe links ins Argument (rekursiv):
tryNameBeta (App e1 e2) freshvars =
 case tryNameBeta e1 freshvars of
  Nothing -> Nothing
  Just (e1',vars) -> (Just ((App e1' e2), vars))

-- Andere Faelle: Keine Reduktion m"oglich:
tryNameBeta _ vars = Nothing
```

Implementierung eines Interpreters für den Lambda-Kalkül (5)

Auswerten, solange wie möglich:

```
tryName e vars = case tryNameBeta e vars of
                   Nothing -> e
                   Just (e',vars') -> tryName e' vars'


reduceName e = let (e',v') = rename e fresh
               in  tryName e' v'
      where fresh = ["x_" ++ show i | i <- [1..]]
```

Implementierung eines Interpreters für den Lambda-Kalkül (6)

Beispielausdrücke:

```
example_id  = (Lambda "v" (Var "v"))
example_k   = (Lambda "x" (Lambda "y" (Var "x")))
```

Beispielaufrufe:

```
*Main> reduceName example_id
Lambda "x_1" (Var "x_1")
*Main> reduceName (App example_id example_id)
Lambda "x_3" (Var "x_3")
*Main> reduceName (App example_k example_id)
Lambda "x_2" (Lambda "x_4" (Var "x_4"))
```

Call-by-Value-Reduktion

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

Call-by-value Beta-Reduktion

$$(\beta_{cbv}) \qquad (\lambda x.s)\ v \to s[v/x], \text{ wobei } v \text{ Abstraktion oder Variable}$$

Definition

CBV-Reduktionskontexte E werden erzeugt durch $\mathbf{ECtxt}$:

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

Wenn $r_1 \xrightarrow{\beta_{cbv}} r_2$ und E ein CBV-Reduktionskontext ist,
dann ist $E[r_1] \xrightarrow{value} E[r_2]$ eine Call-by-Value-Reduktion

CBV-Redexsuche mit Markierungsalgorithmus

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

CBV-Redexsuche mit Markierungsalgorithmus

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

CBV-Redexsuche mit Markierungsalgorithmus

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

CBV-Redexsuche mit Markierungsalgorithmus

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

CBV-Redexsuche mit Markierungsalgorithmus

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

CBV-Redexsuche mit Markierungsalgorithmus

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

CBV-Redexsuche mit Markierungsalgorithmus

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

CBV-Redexsuche mit Markierungsalgorithmus

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

CBV-Redexsuche mit Markierungsalgorithmus

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

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

Beispiel

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

Eigenschaften

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

Definition

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

Eigenschaften

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

Definition

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

- Es gilt: $s{\downarrow_{value}} \implies s{\downarrow}$.
- Die Umkehrung gilt nicht!

Eigenschaften

Vorteile der Call-by-Value-Reihenfolge:

- Tlw. besseres Platzverhalten
- Auswertungsreihenfolge liegt fest (im Syntaxbaum von f); bzw. ist vorhersagbar.
  Bei CBV: $f\ s_1\ s_2\ s_3$: immer zuerst s_1, dann s_2, dann s_3, dann $(f\ s_1\ s_2\ s_3)$
  Bei CBN: 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 unter CBV:
  Seiteneffekte können unter CBV direkt eingebaut werden

In Haskell: seq

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

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

- in Call-by-Value-Reduktion: ist seq kodierbar, aber unnötig
- in Call-by-Name-Reduktion: seq kann nicht im Lambda-Kalkül kodiert werden!

Beispiele

- $\Omega := (\lambda x.x\ x)\ (\lambda x.x\ x)$.
- $\Omega \xrightarrow{name} \Omega$. Daraus folgt: $\Omega\Uparrow$
- $\Omega \xrightarrow{value} \Omega$. Daraus folgt: $\Omega\Uparrow_{value}$.
- $t := ((\lambda x.(\lambda y.y))\ \Omega)$.
- $t \xrightarrow{name} \lambda y.y$, d.h. $t\downarrow$.
- $t \xrightarrow{value} t$, also $t\Uparrow_{value}$, denn die Call-by-Value-Reduktion muss zunächst das Argument Ω auswerten.

Verzögerte Auswertung

- Sprechweisen: Verzögerte Auswertung, call-by-need, nicht-strikt, lazy, Sharing
- Optimierung der Call-by-Name-Reduktion

Call-by-need Lambda-Kalkül mit `let` – Syntax:

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

- Nicht-rekursives `let`: in `let` $x = s$ `in` t muss gelten $x \notin FV(s)$
- Haskell verwendet rekursives `let`!

Call-by-Need Lambda Kalkül - Auswertung (1)

Reduktionskontexte $\mathbf{R}_{need}$:

$$\begin{array}{rcl} \mathbf{R}_{need} & ::= & \mathbf{LR}[\mathbf{A}] \mid \mathbf{LR}[\texttt{let } x = \mathbf{A} \texttt{ in } \mathbf{R}_{need}[x]] \\ \mathbf{A} & ::= & [\cdot] \mid (\mathbf{A}\ \mathbf{Expr}) \\ \mathbf{LR} & ::= & [\cdot] \mid \texttt{let } V = \mathbf{Expr} \texttt{ in } \mathbf{LR} \end{array}$$

- $\mathbf{A} \mathrel{\widehat{=}}$ links in die Applikation
- $\mathbf{LR} \mathrel{\widehat{=}}$ Rechts ins `let`

Call-by-Need Lambda Kalkül - Auswertung (2)

Verzögerter Auswertungsschritt $\xrightarrow{need}$, definiert durch 4 Regeln:

$$(lbeta)\ R_{need}[(\lambda x.s)\ t] \to R_{need}[\texttt{let}\ x = t\ \texttt{in}\ s]$$

$$(cp)\quad LR[\texttt{let}\ x = \lambda y.s\ \texttt{in}\ R_{need}[x]] \to LR[\texttt{let}\ x = \lambda y.s\ \texttt{in}\ R_{need}[\lambda y.s]]$$

$$(llet)\quad LR[\texttt{let}\ x = (\texttt{let}\ y = s\ \texttt{in}\ t)\ \texttt{in}\ R_{need}[x]] \to LR[\texttt{let}\ y = s\ \texttt{in}\ (\texttt{let}\ x = t\ \texttt{in}\ R_{need}[x])]$$

$$(lapp)\quad R_{need}[(\texttt{let}\ x = s\ \texttt{in}\ t)\ r] \to R_{need}[\texttt{let}\ x = s\ \texttt{in}\ (t\ r)]$$

- $(lbeta)$ und (cp) statt (β),
- $(lapp)$ und $(llet)$ zum let-Verschieben
- Die Regeln gehen davon aus, dass die Ausdrücke vorher die DVC erfüllen.

Markierungsalgorithmus zur Redexsuche (1)

- Markierungen: $\star, \diamond, \circledcirc$
- $\star \vee \diamond$ meint $\star$ oder $\diamond$
- Für Ausdruck s starte mit $s^{\star}$.

Verschiebe-Regeln:

(1) $(\texttt{let}\ x = s\ \texttt{in}\ t)^{\star} \quad \Rightarrow \quad (\texttt{let}\ x = s\ \texttt{in}\ t^{\star})$

(2) $(\texttt{let}\ x = C_1[y^{\diamond}]\ \texttt{in}\ C_2[x^{\circledcirc}]) \quad \Rightarrow \quad (\texttt{let}\ x = C_1[y^{\diamond}]\ \texttt{in}\ C_2[x])$

(3) $(\texttt{let}\ x = s\ \texttt{in}\ C[x^{\star \vee \diamond}]) \quad \Rightarrow \quad (\texttt{let}\ x = s^{\diamond}\ \texttt{in}\ C[x^{\circledcirc}])$

(4) $(s\ t)^{\star \vee \diamond} \quad \Rightarrow \quad (s^{\diamond}\ t)$

dabei immer (2) statt (3) anwenden falls möglich

Markierungsalgorithmus zur Redexsuche (2)

Reduktion nach Markierung:

$(lbeta)$ $((\lambda x.s)^{\diamond}\ t) \to \texttt{let}\ x = t\ \texttt{in}\ s$

(cp) $\texttt{let}\ x = (\lambda y.s)^{\diamond}\ \texttt{in}\ C[x^{\circledcirc}] \to \texttt{let}\ x = \lambda y.s\ \texttt{in}\ C[\lambda y.s]$

$(llet)$ $\texttt{let}\ x = (\texttt{let}\ y = s\ \texttt{in}\ t)^{\diamond}\ \texttt{in}\ C[x^{\circledcirc}]$
$\to \texttt{let}\ y = s\ \texttt{in}\ (\texttt{let}\ x = t\ \texttt{in}\ C[x])]$

$(lapp)$ $((\texttt{let}\ x = s\ \texttt{in}\ t)^{\diamond}\ r) \to \texttt{let}\ x = s\ \texttt{in}\ (t\ r)$

Beispiel: Call-by-Need-Reduktion

$$(\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ ((\lambda y.y)\ x))^{\star}$$

Beispiel: Call-by-Need-Reduktion

$$(\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ ((\lambda y.y)\ x)^{\star})$$

Beispiel: Call-by-Need-Reduktion

$$(\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ ((\lambda y.y)^{\diamond}\ x))$$

Beispiel: Call-by-Need-Reduktion

$$
\begin{array}{rl}
 & (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ ((\lambda y.y)^{\diamond}\ x)) \\
\xrightarrow{need,lbeta} & (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y))^{\star}
\end{array}
$$

Beispiel: Call-by-Need-Reduktion

$$
\begin{aligned}
& (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ ((\lambda y.y)^{\diamond}\ x)) \\
\xrightarrow{need,lbeta} & (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y)^{\star})
\end{aligned}
$$

Beispiel: Call-by-Need-Reduktion

$$
\begin{aligned}
& (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ ((\lambda y.y)^{\diamond}\ x)) \\
\xrightarrow{need,lbeta} & (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y^{\star}))
\end{aligned}
$$

Beispiel: Call-by-Need-Reduktion

$$
\begin{aligned}
& (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ ((\lambda y.y)^{\diamond}\ x)) \\
\xrightarrow{need,lbeta} & (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = x^{\diamond}\ \texttt{in}\ y^{\circledcirc}))
\end{aligned}
$$

Beispiel: Call-by-Need-Reduktion

$$
\begin{aligned}
& (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ ((\lambda y.y)^{\diamond}\ x)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = x^{\diamond}\ \texttt{in}\ y))
\end{aligned}
$$

Beispiel: Call-by-Need-Reduktion

$$
\begin{aligned}
& (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ ((\lambda y.y)^{\diamond}\ x)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = ((\lambda u.u)\ (\lambda w.w))^{\diamond}\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y))
\end{aligned}
$$

Beispiel: Call-by-Need-Reduktion

$$
\begin{aligned}
& (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ ((\lambda y.y)^{\diamond}\ x)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = ((\lambda u.u)^{\diamond}\ (\lambda w.w))\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y))
\end{aligned}
$$

Beispiel: Call-by-Need-Reduktion

$$
\begin{aligned}
& (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ ((\lambda y.y)^{\diamond}\ x)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = ((\lambda u.u)^{\diamond}\ (\lambda w.w))\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = (\texttt{let}\ u = \lambda w.w\ \texttt{in}\ u)\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y))^{\star}
\end{aligned}
$$

Beispiel: Call-by-Need-Reduktion

$$
\begin{aligned}
& (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ ((\lambda y.y)^{\diamond}\ x)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = ((\lambda u.u)^{\diamond}\ (\lambda w.w))\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = (\texttt{let}\ u = \lambda w.w\ \texttt{in}\ u)\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y)^{\star})
\end{aligned}
$$

Beispiel: Call-by-Need-Reduktion

$$
\begin{aligned}
& (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ ((\lambda y.y)^{\diamond}\ x)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = ((\lambda u.u)^{\diamond}\ (\lambda w.w))\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = (\texttt{let}\ u = \lambda w.w\ \texttt{in}\ u)\ \texttt{in}\ (\texttt{let}\ y = x^{\diamond}\ \texttt{in}\ y^{\circledcirc}))
\end{aligned}
$$

Beispiel: Call-by-Need-Reduktion

$$
\begin{aligned}
& (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ ((\lambda y.y)^{\diamond}\ x)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = ((\lambda u.u)^{\diamond}\ (\lambda w.w))\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = (\texttt{let}\ u = \lambda w.w\ \texttt{in}\ u)\ \texttt{in}\ (\texttt{let}\ y = x^{\diamond}\ \texttt{in}\ y))
\end{aligned}
$$

Beispiel: Call-by-Need-Reduktion

$$
\begin{aligned}
& (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ ((\lambda y.y)^{\diamond}\ x)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = ((\lambda u.u)^{\diamond}\ (\lambda w.w))\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = (\texttt{let}\ u = \lambda w.w\ \texttt{in}\ u)^{\diamond}\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y))
\end{aligned}
$$

Beispiel: Call-by-Need-Reduktion

$$
\begin{aligned}
& (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ ((\lambda y.y)^{\diamond}\ x)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = ((\lambda u.u)^{\diamond}\ (\lambda w.w))\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = (\texttt{let}\ u = \lambda w.w\ \texttt{in}\ u)^{\diamond}\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,llet}\ & (\texttt{let}\ u = \lambda w.w\ \texttt{in}\ (\texttt{let}\ x = u\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y)))^{\star}
\end{aligned}
$$

Beispiel: Call-by-Need-Reduktion

$$
\begin{aligned}
& (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ ((\lambda y.y)^{\diamond}\ x)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = ((\lambda u.u)^{\diamond}\ (\lambda w.w))\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = (\texttt{let}\ u = \lambda w.w\ \texttt{in}\ u)^{\diamond}\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,llet}\ & (\texttt{let}\ u = \lambda w.w\ \texttt{in}\ (\texttt{let}\ x = u\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y))^{\star})
\end{aligned}
$$

Beispiel: Call-by-Need-Reduktion

$$
\begin{aligned}
& (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ ((\lambda y.y)^{\diamond}\ x)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = ((\lambda u.u)^{\diamond}\ (\lambda w.w))\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = (\texttt{let}\ u = \lambda w.w\ \texttt{in}\ u)^{\diamond}\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,llet}\ & (\texttt{let}\ u = \lambda w.w\ \texttt{in}\ (\texttt{let}\ x = u\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y)^{\star}))
\end{aligned}
$$

Beispiel: Call-by-Need-Reduktion

$$
\begin{aligned}
& (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ ((\lambda y.y)^{\diamond}\ x)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = ((\lambda u.u)^{\diamond}\ (\lambda w.w))\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = (\texttt{let}\ u = \lambda w.w\ \texttt{in}\ u)^{\diamond}\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,llet}\ & (\texttt{let}\ u = \lambda w.w\ \texttt{in}\ (\texttt{let}\ x = u\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y^{\star})))
\end{aligned}
$$

Beispiel: Call-by-Need-Reduktion

$$
\begin{aligned}
& (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ ((\lambda y.y)^{\diamond}\ x)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = ((\lambda u.u)^{\diamond}\ (\lambda w.w))\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = (\texttt{let}\ u = \lambda w.w\ \texttt{in}\ u)^{\diamond}\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,llet}\ & (\texttt{let}\ u = \lambda w.w\ \texttt{in}\ (\texttt{let}\ x = u\ \texttt{in}\ (\texttt{let}\ y = x^{\diamond}\ \texttt{in}\ y^{\circledcirc})))
\end{aligned}
$$

Beispiel: Call-by-Need-Reduktion

$$
\begin{aligned}
& (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ ((\lambda y.y)^{\diamond}\ x)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = ((\lambda u.u)^{\diamond}\ (\lambda w.w))\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = (\texttt{let}\ u = \lambda w.w\ \texttt{in}\ u)^{\diamond}\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,llet}\ & (\texttt{let}\ u = \lambda w.w\ \texttt{in}\ (\texttt{let}\ x = u\ \texttt{in}\ (\texttt{let}\ y = x^{\diamond}\ \texttt{in}\ y)))
\end{aligned}
$$

Beispiel: Call-by-Need-Reduktion

$$
\begin{aligned}
& (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ ((\lambda y.y)^{\diamond}\ x)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = ((\lambda u.u)^{\diamond}\ (\lambda w.w))\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = (\texttt{let}\ u = \lambda w.w\ \texttt{in}\ u)^{\diamond}\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,llet}\ & (\texttt{let}\ u = \lambda w.w\ \texttt{in}\ (\texttt{let}\ x = u^{\diamond}\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)))
\end{aligned}
$$

Beispiel: Call-by-Need-Reduktion

$$
\begin{aligned}
& (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ ((\lambda y.y)^{\diamond}\ x)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = ((\lambda u.u)^{\diamond}\ (\lambda w.w))\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = (\texttt{let}\ u = \lambda w.w\ \texttt{in}\ u)^{\diamond}\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,llet}\ & (\texttt{let}\ u = \lambda w.w\ \texttt{in}\ (\texttt{let}\ x = u^{\diamond}\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y)))
\end{aligned}
$$

Beispiel: Call-by-Need-Reduktion

$$
\begin{aligned}
& (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ ((\lambda y.y)^{\diamond}\ x)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = ((\lambda u.u)^{\diamond}\ (\lambda w.w))\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = (\texttt{let}\ u = \lambda w.w\ \texttt{in}\ u)^{\diamond}\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,llet}\ & (\texttt{let}\ u = (\lambda w.w)^{\diamond}\ \texttt{in}\ (\texttt{let}\ x = u^{\circledcirc}\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y)))
\end{aligned}
$$

Beispiel: Call-by-Need-Reduktion

$$
\begin{aligned}
& (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ ((\lambda y.y)^{\diamond}\ x)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = ((\lambda u.u)^{\diamond}\ (\lambda w.w))\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = (\texttt{let}\ u = \lambda w.w\ \texttt{in}\ u)^{\diamond}\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,llet}\ & (\texttt{let}\ u = (\lambda w.w)^{\diamond}\ \texttt{in}\ (\texttt{let}\ x = u^{\circledcirc}\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y))) \\
\xrightarrow{need,cp}\ & (\texttt{let}\ u = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ x = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y)))^{\star}
\end{aligned}
$$

Beispiel: Call-by-Need-Reduktion

$$
\begin{aligned}
& (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ ((\lambda y.y)^{\diamond}\ x)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = ((\lambda u.u)^{\diamond}\ (\lambda w.w))\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = (\texttt{let}\ u = \lambda w.w\ \texttt{in}\ u)^{\diamond}\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,llet}\ & (\texttt{let}\ u = (\lambda w.w)^{\diamond}\ \texttt{in}\ (\texttt{let}\ x = u^{\circledcirc}\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y))) \\
\xrightarrow{need,cp}\ & (\texttt{let}\ u = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ x = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y))^{\star})
\end{aligned}
$$

Beispiel: Call-by-Need-Reduktion

$$
\begin{aligned}
& (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ ((\lambda y.y)^{\diamond}\ x)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = ((\lambda u.u)^{\diamond}\ (\lambda w.w))\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = (\texttt{let}\ u = \lambda w.w\ \texttt{in}\ u)^{\diamond}\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,llet}\ & (\texttt{let}\ u = (\lambda w.w)^{\diamond}\ \texttt{in}\ (\texttt{let}\ x = u^{\circledcirc}\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y))) \\
\xrightarrow{need,cp}\ & (\texttt{let}\ u = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ x = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y)^{\star}))
\end{aligned}
$$

Beispiel: Call-by-Need-Reduktion

$$
\begin{aligned}
& (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ ((\lambda y.y)^{\diamond}\ x)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = ((\lambda u.u)^{\diamond}\ (\lambda w.w))\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = (\texttt{let}\ u = \lambda w.w\ \texttt{in}\ u)^{\diamond}\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,llet}\ & (\texttt{let}\ u = (\lambda w.w)^{\diamond}\ \texttt{in}\ (\texttt{let}\ x = u^{\circledcirc}\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y))) \\
\xrightarrow{need,cp}\ & (\texttt{let}\ u = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ x = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y^{\star})))
\end{aligned}
$$

Beispiel: Call-by-Need-Reduktion

$$
\begin{aligned}
& (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ ((\lambda y.y)^{\diamond}\ x)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = ((\lambda u.u)^{\diamond}\ (\lambda w.w))\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = (\texttt{let}\ u = \lambda w.w\ \texttt{in}\ u)^{\diamond}\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,llet}\ & (\texttt{let}\ u = (\lambda w.w)^{\diamond}\ \texttt{in}\ (\texttt{let}\ x = u^{\circledcirc}\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y))) \\
\xrightarrow{need,cp}\ & (\texttt{let}\ u = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ x = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = x^{\diamond}\ \texttt{in}\ y^{\circledcirc})))
\end{aligned}
$$

Beispiel: Call-by-Need-Reduktion

$$
\begin{aligned}
& (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ ((\lambda y.y)^{\diamond}\ x)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = ((\lambda u.u)^{\diamond}\ (\lambda w.w))\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = (\texttt{let}\ u = \lambda w.w\ \texttt{in}\ u)^{\diamond}\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,llet}\ & (\texttt{let}\ u = (\lambda w.w)^{\diamond}\ \texttt{in}\ (\texttt{let}\ x = u^{\circledcirc}\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y))) \\
\xrightarrow{need,cp}\ & (\texttt{let}\ u = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ x = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = x^{\diamond}\ \texttt{in}\ y)))
\end{aligned}
$$

Beispiel: Call-by-Need-Reduktion

$$
\begin{aligned}
& (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ ((\lambda y.y)^{\diamond}\ x)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = ((\lambda u.u)^{\diamond}\ (\lambda w.w))\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = (\texttt{let}\ u = \lambda w.w\ \texttt{in}\ u)^{\diamond}\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,llet}\ & (\texttt{let}\ u = (\lambda w.w)^{\diamond}\ \texttt{in}\ (\texttt{let}\ x = u^{\circledcirc}\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y))) \\
\xrightarrow{need,cp}\ & (\texttt{let}\ u = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ x = (\lambda w.w)^{\diamond}\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)))
\end{aligned}
$$

Beispiel: Call-by-Need-Reduktion

$$
\begin{aligned}
& (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ ((\lambda y.y)^{\diamond}\ x)) \\
\xrightarrow{need,lbeta} & (\texttt{let}\ x = ((\lambda u.u)^{\diamond}\ (\lambda w.w))\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,lbeta} & (\texttt{let}\ x = (\texttt{let}\ u = \lambda w.w\ \texttt{in}\ u)^{\diamond}\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,llet} & (\texttt{let}\ u = (\lambda w.w)^{\diamond}\ \texttt{in}\ (\texttt{let}\ x = u^{\circledcirc}\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y))) \\
\xrightarrow{need,cp} & (\texttt{let}\ u = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ x = (\lambda w.w)^{\diamond}\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y))) \\
\xrightarrow{need,cp} & (\texttt{let}\ u = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ x = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = (\lambda w.w)\ \texttt{in}\ y)))^{\star}
\end{aligned}
$$

Beispiel: Call-by-Need-Reduktion

$$
\begin{aligned}
& (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ ((\lambda y.y)^{\diamond}\ x)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = ((\lambda u.u)^{\diamond}\ (\lambda w.w))\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = (\texttt{let}\ u = \lambda w.w\ \texttt{in}\ u)^{\diamond}\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,llet}\ & (\texttt{let}\ u = (\lambda w.w)^{\diamond}\ \texttt{in}\ (\texttt{let}\ x = u^{\circledcirc}\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y))) \\
\xrightarrow{need,cp}\ & (\texttt{let}\ u = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ x = (\lambda w.w)^{\diamond}\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y))) \\
\xrightarrow{need,cp}\ & (\texttt{let}\ u = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ x = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = (\lambda w.w)\ \texttt{in}\ y))^{\star})
\end{aligned}
$$

Beispiel: Call-by-Need-Reduktion

$$
\begin{aligned}
& (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ ((\lambda y.y)^{\diamond}\ x)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = ((\lambda u.u)^{\diamond}\ (\lambda w.w))\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = (\texttt{let}\ u = \lambda w.w\ \texttt{in}\ u)^{\diamond}\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,llet}\ & (\texttt{let}\ u = (\lambda w.w)^{\diamond}\ \texttt{in}\ (\texttt{let}\ x = u^{\circledcirc}\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y))) \\
\xrightarrow{need,cp}\ & (\texttt{let}\ u = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ x = (\lambda w.w)^{\diamond}\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y))) \\
\xrightarrow{need,cp}\ & (\texttt{let}\ u = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ x = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = (\lambda w.w)\ \texttt{in}\ y)^{\star}))
\end{aligned}
$$

Beispiel: Call-by-Need-Reduktion

$$
\begin{aligned}
& (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ ((\lambda y.y)^{\diamond}\ x)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = ((\lambda u.u)^{\diamond}\ (\lambda w.w))\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = (\texttt{let}\ u = \lambda w.w\ \texttt{in}\ u)^{\diamond}\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,llet}\ & (\texttt{let}\ u = (\lambda w.w)^{\diamond}\ \texttt{in}\ (\texttt{let}\ x = u^{\circledcirc}\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y))) \\
\xrightarrow{need,cp}\ & (\texttt{let}\ u = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ x = (\lambda w.w)^{\diamond}\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y))) \\
\xrightarrow{need,cp}\ & (\texttt{let}\ u = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ x = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = (\lambda w.w)\ \texttt{in}\ y^{\star})))
\end{aligned}
$$

Beispiel: Call-by-Need-Reduktion

$$
\begin{aligned}
& (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ ((\lambda y.y)^{\diamond}\ x)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = ((\lambda u.u)^{\diamond}\ (\lambda w.w))\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = (\texttt{let}\ u = \lambda w.w\ \texttt{in}\ u)^{\diamond}\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,llet}\ & (\texttt{let}\ u = (\lambda w.w)^{\diamond}\ \texttt{in}\ (\texttt{let}\ x = u^{\circledcirc}\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y))) \\
\xrightarrow{need,cp}\ & (\texttt{let}\ u = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ x = (\lambda w.w)^{\diamond}\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y))) \\
\xrightarrow{need,cp}\ & (\texttt{let}\ u = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ x = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = (\lambda w.w)^{\diamond}\ \texttt{in}\ y^{\circledcirc})))
\end{aligned}
$$

Beispiel: Call-by-Need-Reduktion

$$
\begin{aligned}
& (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ ((\lambda y.y)^{\diamond}\ x)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = ((\lambda u.u)^{\diamond}\ (\lambda w.w))\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = (\texttt{let}\ u = \lambda w.w\ \texttt{in}\ u)^{\diamond}\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,llet}\ & (\texttt{let}\ u = (\lambda w.w)^{\diamond}\ \texttt{in}\ (\texttt{let}\ x = u^{\circledcirc}\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y))) \\
\xrightarrow{need,cp}\ & (\texttt{let}\ u = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ x = (\lambda w.w)^{\diamond}\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y))) \\
\xrightarrow{need,cp}\ & (\texttt{let}\ u = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ x = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = (\lambda w.w)^{\diamond}\ \texttt{in}\ y^{\circledcirc}))) \\
\xrightarrow{need,cp}\ & (\texttt{let}\ u = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ x = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = (\lambda w.w)\ \texttt{in}\ (\lambda w.w))))
\end{aligned}
$$

Beispiel: Call-by-Need-Reduktion

$$
\begin{aligned}
& (\texttt{let}\ x = (\lambda u.u)\ (\lambda w.w)\ \texttt{in}\ ((\lambda y.y)^{\diamond}\ x)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = ((\lambda u.u)^{\diamond}\ (\lambda w.w))\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,lbeta}\ & (\texttt{let}\ x = (\texttt{let}\ u = \lambda w.w\ \texttt{in}\ u)^{\diamond}\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y)) \\
\xrightarrow{need,llet}\ & (\texttt{let}\ u = (\lambda w.w)^{\diamond}\ \texttt{in}\ (\texttt{let}\ x = u^{\circledcirc}\ \texttt{in}\ (\texttt{let}\ y = x\ \texttt{in}\ y))) \\
\xrightarrow{need,cp}\ & (\texttt{let}\ u = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ x = (\lambda w.w)^{\diamond}\ \texttt{in}\ (\texttt{let}\ y = x^{\circledcirc}\ \texttt{in}\ y))) \\
\xrightarrow{need,cp}\ & (\texttt{let}\ u = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ x = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = (\lambda w.w)^{\diamond}\ \texttt{in}\ y^{\circledcirc}))) \\
\xrightarrow{need,cp}\ & (\texttt{let}\ u = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ x = (\lambda w.w)\ \texttt{in}\ (\texttt{let}\ y = (\lambda w.w)\ \texttt{in}\ (\lambda w.w))))
\end{aligned}
$$

- Der letzte Ausdruck ist eine call-by-need FWHNF
- Call-by-need FWHNF: Ausdruck der Form $LR[\lambda x.s]$, d.h.

$$
\begin{array}{l}
\texttt{let}\ x_1 = s_1\ \texttt{in} \\
\quad (\texttt{let}\ x_2 = s_2\ \texttt{in} \\
\qquad (\ldots \\
\qquad\quad (\texttt{let}\ x_n = s_n\ \texttt{in}\ \lambda x.s)))
\end{array}
$$

Konvergenz und Eigenschaften

Definition

Ein Ausdruck s call-by-need konvergiert (geschrieben als $s{\downarrow_{need}}$), gdw. er mit einer Folge von $\xrightarrow{need}$-Reduktionen in eine FWHNF überführt werden kann, d.h.

$$s{\downarrow_{need}} \iff \exists \text{ FWHNF } v : s \xrightarrow{need,*} v$$

Satz

Sei s ein (`let`-freier) Ausdruck, dann gilt $s{\downarrow} \iff s{\downarrow_{need}}$.

Gleichheit

Kalküle bisher:

- Call-by-Name Lambda-Kalkül: Ausdrücke, $\xrightarrow{name}$, $\downarrow$
- Call-by-Value Lambda-Kalkül: Ausdrücke, $\xrightarrow{value}$, $\downarrow_{value}$
- (Call-by-Need Lambda-Kalkül Ausdrücke, $\xrightarrow{need}$, $\downarrow_{need}$)

D.h. Syntax + Operationale Semantik.

Es fehlt:

- Begriff: Wann sind zwei Ausdrücke gleich(wertig)?
- Insbesondere: Wann darf ein Compiler einen Ausdruck durch einen anderen ersetzen?

Gleichheit (2)

- Leibnizsches Prinzip: Zwei Dinge sind gleich, wenn sie die gleichen Eigenschaften haben, bzgl. aller Eigenschaften.
- Für Kalküle: Zwei Ausdrücke s, t sind gleich, wenn man sie nicht unterscheiden kann, 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 (Konvergenz)
  (ergibt das gleiche wie Reduktion auf gleiche "Werte").

Gleichheit (2)

- Leibnizsches Prinzip: Zwei Dinge sind gleich, wenn sie die gleichen Eigenschaften haben, bzgl. aller Eigenschaften.
- Für Kalküle: Zwei Ausdrücke s, t sind gleich, wenn man sie nicht unterscheiden kann, 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 (Konvergenz)
  (ergibt das gleiche wie Reduktion auf gleiche "Werte").

Gleichheit (2)

- Leibnizsches Prinzip: Zwei Dinge sind gleich, wenn sie die gleichen Eigenschaften haben, bzgl. aller Eigenschaften.
- Für Kalküle: Zwei Ausdrücke s, t sind gleich, wenn man sie nicht unterscheiden kann, egal in welchem Kontext man sie benutzt.
- Formaler: s,t sind gleich, wenn für alle Kontexte C gilt:

$$C[s] \text{ und } C[t] \text{ verhalten sich gleich.}$$

- **Verhalten** muss noch definiert werden. Für deterministische Sprachen reicht die Beobachtung der Terminierung (Konvergenz)
(ergibt das gleiche wie Reduktion auf gleiche "Werte").

Gleichheit (2)

- Leibnizsches Prinzip: Zwei Dinge sind gleich, wenn sie die gleichen Eigenschaften haben, bzgl. aller Eigenschaften.
- Für Kalküle: Zwei Ausdrücke s, t sind gleich, wenn man sie nicht unterscheiden kann, egal in welchem Kontext man sie benutzt.
- Formaler: s,t sind gleich, wenn für alle Kontexte C gilt:

$$C[s] \text{ und } C[t] \text{ verhalten sich gleich.}$$

- **Verhalten** muss noch definiert werden. Für deterministische Sprachen reicht die Beobachtung der Terminierung (Konvergenz)
(ergibt das gleiche wie Reduktion auf gleiche "Werte").

Gleichheit (3)

Kontextuelle Approximation und Gleichheit

Call-by-Name Lambda-Kalkül:

- $s \leq_c t$ gdw. $\forall C : C[s]{\downarrow} \implies C[t]{\downarrow}$

Gleichheit (3)

Kontextuelle Approximation und Gleichheit

Call-by-Name Lambda-Kalkül:

- $s \leq_c t$ gdw. $\forall C : C[s]{\downarrow} \implies C[t]{\downarrow}$
- $s \sim_c t$ 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. $\forall C : C[s]\downarrow \implies C[t]\downarrow$
- $s \sim_c t$ gdw. $s \leq_c t$ und $t \leq_c s$

Call-by-Value Lambda-Kalkül:

- $s \leq_{c,value} t$ gdw. $\forall C$ mit $FV(C[s]\ C[t]) = \emptyset$: $C[s]\downarrow_{value} \implies C[t]\downarrow_{value}$.
- $s \sim_{c,value} t$ gdw. $s \leq_{c,value} t$ und $t \leq_{c,value} s$

Gleichheit (4)

- $\sim_c$ und $\sim_{value}$ sind Kongruenzen
- Kongruenz = Äquivalenzrelation + kompatibel mit Kontexten, d.h. $s \sim t \implies C[s] \sim C[t]$.
- Gleichheit beweisen i.a. schwer, widerlegen i.a. einfach.

Gleichheit (4)

- $\sim_c$ und $\sim_{value}$ sind Kongruenzen
- Kongruenz = Äquivalenzrelation + kompatibel mit Kontexten, d.h. $s \sim t \implies C[s] \sim C[t]$.
- Gleichheit beweisen i.a. schwer, widerlegen i.a. einfach.

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

- geschlossen natürlich, da nur geschlossene Ausdrücke „Programme“ sind
- $\sim_c$: gleiche Relation, egal ob alle oder schließende Kontexte
- $\sim_{c,value}$: Relation ändert sich, wenn man alle Kontexte verwendet!

Gleichheit (5)

Beispiele für Gleichheiten:

- $(\beta) \subseteq \sim_c$
- $(\beta_{cbv}) \subseteq \sim_{c,value}$ aber $(\beta) \not\subseteq \sim_{c,value}$
- $\sim_c \not\subseteq \sim_{c,value}$ und $\sim_{c,value} \not\subseteq \sim_c$

Lambda-Kalkül und Haskell

Der Lambda-Kalkül ist als Kernsprache für Haskell eher ungeeignet:

Lambda-Kalkül und Haskell

Der Lambda-Kalkül ist als Kernsprache für Haskell eher ungeeignet:

- Keine echten Daten:
  Zahlen, Boolesche Werte, komplexe Datenstrukturen fehlen im Lambda-Kalkül.

Lambda-Kalkül und Haskell

Der Lambda-Kalkül ist als Kernsprache für Haskell eher ungeeignet:

- Keine echten Daten:
  Zahlen, Boolesche Werte, komplexe Datenstrukturen fehlen im Lambda-Kalkül.
  Ausweg mittels Church-Kodierung:

$$\begin{array}{lll} \text{z.B. } \textit{true} & = \lambda x, y.x \\ \textit{false} & = \lambda x, y.y \\ \textit{if-then-else} & = \lambda b, x_1, x_2.b\ x_1\ x_2 \end{array}$$

if-then-else true $e_1\ e_2 = (\lambda b, x_1, x_2.b\ x_1\ x_2)\ (\lambda x, y.x)\ e_1\ e_2 \xrightarrow{name,\beta,3} (\lambda x, y.x)\ e_1\ e_2 \xrightarrow{name,\beta,2} e_1$

if-then-else false $e_1\ e_2 = (\lambda b, x_1, x_2.b\ x_1\ x_2)\ (\lambda x, y.y)\ e_1\ e_2 \xrightarrow{name,\beta,3} (\lambda x, y.y)\ e_1\ e_2 \xrightarrow{name,\beta,2} e_2$

Aber: Kompliziert, Daten + Funktionen nicht unterscheidbar, Typisierung passt nicht

Lambda-Kalkül und Haskell

Der Lambda-Kalkül ist als Kernsprache für Haskell eher ungeeignet:

- Keine echten Daten:
  Zahlen, Boolesche Werte, komplexe Datenstrukturen fehlen im Lambda-Kalkül.
  Ausweg mittels Church-Kodierung:

$$\begin{array}{lll} \text{z.B. } \textit{true} & = \lambda x, y.x \\ \textit{false} & = \lambda x, y.y \\ \textit{if-then-else} & = \lambda b, x_1, x_2.b\ x_1\ x_2 \end{array}$$

if-then-else true $e_1\ e_2 = (\lambda b, x_1, x_2.b\ x_1\ x_2)\ (\lambda x, y.x)\ e_1\ e_2 \xrightarrow{name,\beta,3} (\lambda x, y.x)\ e_1\ e_2 \xrightarrow{name,\beta,2} e_1$

if-then-else false $e_1\ e_2 = (\lambda b, x_1, x_2.b\ x_1\ x_2)\ (\lambda x, y.y)\ e_1\ e_2 \xrightarrow{name,\beta,3} (\lambda x, y.y)\ e_1\ e_2 \xrightarrow{name,\beta,2} e_2$

  Aber: Kompliziert, Daten + Funktionen nicht unterscheidbar, Typisierung passt nicht

- Typisierung fehlt Haskell ist polymorph getypt.

Lambda-Kalkül und Haskell

Der Lambda-Kalkül ist als Kernsprache für Haskell eher ungeeignet:

- Keine echten Daten:
  Zahlen, Boolesche Werte, komplexe Datenstrukturen fehlen im Lambda-Kalkül.
  Ausweg mittels Church-Kodierung:

$$\begin{array}{lll} \text{z.B. } \textit{true} & = \lambda x, y.x \\ \textit{false} & = \lambda x, y.y \\ \textit{if-then-else} & = \lambda b, x_1, x_2.b\ x_1\ x_2 \end{array}$$

$\textit{if-then-else true}\ e_1\ e_2 = (\lambda b, x_1, x_2.b\ x_1\ x_2)\ (\lambda x, y.x)\ e_1\ e_2 \xrightarrow{name,\beta,3} (\lambda x, y.x)\ e_1\ e_2 \xrightarrow{name,\beta,2} e_1$

$\textit{if-then-else false}\ e_1\ e_2 = (\lambda b, x_1, x_2.b\ x_1\ x_2)\ (\lambda x, y.y)\ e_1\ e_2 \xrightarrow{name,\beta,3} (\lambda x, y.y)\ e_1\ e_2 \xrightarrow{name,\beta,2} e_2$

  Aber: Kompliziert, Daten + Funktionen nicht unterscheidbar, Typisierung passt nicht

- Typisierung fehlt Haskell ist polymorph getypt.
- Kein seq: In Haskell ist seq verfügbar, im Lambda-Kalkül nicht kodierbar.

Lambda-Kalkül und Haskell (2)

- Rekursive Funktionen: Geht nur über Fixpunkt-Kombinatoren:
  Der Ausdruck $Y = \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$
  ist ein Fixpunktkombinator, denn $Y\ f \sim_c f\ (Y\ f)$

  Fakultät als Beispiel:

$$fak = Y\ (\lambda f.\lambda x.\texttt{if}\ x = 0\ \texttt{then}\ 1\ \texttt{else}\ x * (f\ (x-1)))$$

  Aber: Kompliziert und schwer verständlich!
  Bemerkung: Es gibt unendlich viele Fixpunktkombinatoren, u.a. (Klop 2007):

$$\begin{array}{lcl} Y_{Klop} & = & LLLLLLLLLLLLLLLLLLLLLLLLLL \\ L & = & \lambda a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,s,t,u,v,w,x,y,z,r. \\ & & \qquad r\,(this\,is\,a\,fixed\,point\,combinator) \end{array}$$

Kernsprachen für Haskell

- Im folgenden führen wir verschiedene Kernsprachen ein.
- Alle sind Erweiterungen des Lambda-Kalküls
- Bezeichnungen: KFP...
- KFP = Kern einer Funktionalen Programmiersprache

Die Kernsprache KFPT

Die Kernsprache KFPT

- Erweiterung des Lambda-Kalküls um Datentypen (Konstruktoren) und `case`.
- KFPT: T steht für getyptes `case`
- Beachte: KFPT ist nur ganz schwach getypt.

Datentypen

Annahmen:

- Es gibt eine (endliche) Menge von Typen (das sind nur Namen)

Datentypen

Annahmen:

- Es gibt eine (endliche) Menge von Typen (das sind nur Namen)

Beispiele

- Typ `Bool`,
- Typ `List`,

Datentypen

Annahmen:

- Es gibt eine (endliche) Menge von Typen (das sind nur Namen)
- Für jeden Typ gibt es eine endliche Menge von Datenkonstruktoren: Notation: c_i.

Beispiele

- Typ `Bool`,
- Typ `List`,

Datentypen

Annahmen:

- Es gibt eine (endliche) Menge von Typen (das sind nur Namen)
- Für jeden Typ gibt es eine endliche Menge von Datenkonstruktoren: Notation: c_i.

Beispiele

- Typ `Bool`, Datenkonstruktoren: `True` und `False`
- Typ `List`, Datenkonstruktoren: `Nil` und `Cons`

Datentypen

Annahmen:

- Es gibt eine (endliche) Menge von Typen (das sind nur Namen)
- Für jeden Typ gibt es eine endliche Menge von Datenkonstruktoren: Notation: c_i.
- Datenkonstruktoren haben eine Stelligkeit $ar(c_i) \in \mathbb{N}_0$
  ($ar =$ „arity“)

Beispiele

- Typ `Bool`, Datenkonstruktoren: `True` und `False`
- Typ `List`, Datenkonstruktoren: `Nil` und `Cons`

Datentypen

Annahmen:

- Es gibt eine (endliche) Menge von Typen (das sind nur Namen)
- Für jeden Typ gibt es eine endliche Menge von Datenkonstruktoren: Notation: c_i.
- Datenkonstruktoren haben eine Stelligkeit $ar(c_i) \in \mathbb{N}_0$
  ($ar =$ „arity“)

Beispiele

- Typ `Bool`, Datenkonstruktoren: `True` und `False`, $ar(\texttt{True}) = 0 = ar(\texttt{False})$.
- Typ `List`, Datenkonstruktoren: `Nil` und `Cons`,
  $ar(\texttt{Nil}) = 0$ und $ar(\texttt{Cons}) = 2$.

Datentypen

Annahmen:

- Es gibt eine (endliche) Menge von Typen (das sind nur Namen)
- Für jeden Typ gibt es eine endliche Menge von Datenkonstruktoren: Notation: c_i.
- Datenkonstruktoren haben eine Stelligkeit $ar(c_i) \in \mathbb{N}_0$
  ($ar =$ „arity“)

Beispiele

- Typ `Bool`, Datenkonstruktoren: `True` und `False`, $ar(\texttt{True}) = 0 = ar(\texttt{False})$.
- Typ `List`, Datenkonstruktoren: `Nil` und `Cons`,
  $ar(\texttt{Nil}) = 0$ und $ar(\texttt{Cons}) = 2$.

  Haskell-Schreibweise: `[]` für `Nil` und `:` (infix) für `Cons`

Syntax von KFPT

$$
\begin{array}{ll}
\mathbf{Expr} ::= V & \text{(Variable)} \\
\quad \mid \lambda V.\mathbf{Expr} & \text{(Abstraktion)} \\
\quad \mid (\mathbf{Expr}_1\ \mathbf{Expr}_2) & \text{(Anwendung)} \\
\quad \mid & \\
\end{array}
$$

Syntax von KFPT

$$
\begin{array}{lll}
\mathbf{Expr} ::= V & & \text{(Variable)} \\
\quad | \quad \lambda V.\mathbf{Expr} & & \text{(Abstraktion)} \\
\quad | \quad (\mathbf{Expr}_1\ \mathbf{Expr}_2) & & \text{(Anwendung)} \\
\quad | \quad (c_i\ \mathbf{Expr}_1\ \ldots\ \mathbf{Expr}_{ar(c_i)}) & & \text{(Konstruktoranwendung)} \\
\quad | \quad (\mathtt{case}_{\mathtt{Typname}}\ \mathbf{Expr}\ \mathtt{of} & & \text{(case-Ausdruck)} \\
\qquad \{\mathbf{Pat}_1 \to \mathbf{Expr}_1; \ldots; \mathbf{Pat}_n \to \mathbf{Expr}_n\}) & &
\end{array}
$$

$\mathbf{Pat}_i ::= (c_i\ V_1 \ldots V_{ar(c_i)})$ (Pattern für Konstruktor i)

wobei die Variablen V_i alle verschieden sind.

Nebenbedingungen:

- case mit Typ gekennzeichnet,
- $\mathbf{Pat}_i \to \mathbf{Expr}_i$ heißt case-Alternative
- case-Alternativen sind vollständig und disjunkt für den Typ:
  für jeden Konstruktor des Typs kommt genau eine Alternative vor.

Beispiele (1)

Erstes Element einer Liste (head):

$$\lambda xs.\texttt{case}_{\texttt{List}}\ xs\ \texttt{of}\ \{\texttt{Nil} \to \bot; (\texttt{Cons}\ y\ ys) \to y\}$$

Beispiele (1)

Erstes Element einer Liste (head):

$$\lambda xs.\texttt{case}_{\texttt{List}}\ xs\ \texttt{of}\ \{\texttt{Nil} \to \bot; (\texttt{Cons}\ y\ ys) \to y\}$$

Restliste ohne erstes Element (tail):

$$\lambda xs.\texttt{case}_{\texttt{List}}\ xs\ \texttt{of}\ \{\texttt{Nil} \to \bot; (\texttt{Cons}\ y\ ys) \to ys\}$$

$\bot$ repräsentiert Fehler, z.B Ω

Beispiele (1)

Erstes Element einer Liste (head):

$$\lambda xs.\texttt{case}_{\texttt{List}}\ xs\ \texttt{of}\ \{\texttt{Nil} \to \bot; (\texttt{Cons}\ y\ ys) \to y\}$$

Restliste ohne erstes Element (tail):

$$\lambda xs.\texttt{case}_{\texttt{List}}\ xs\ \texttt{of}\ \{\texttt{Nil} \to \bot; (\texttt{Cons}\ y\ ys) \to ys\}$$

$\bot$ repräsentiert Fehler, z.B Ω

Test, ob Liste leer ist (null):

$$\lambda xs.\texttt{case}_{\texttt{List}}\ xs\ \texttt{of}\ \{\texttt{Nil} \to \texttt{True}; (\texttt{Cons}\ y\ ys) \to \texttt{False}\}$$

Beispiele (1)

Erstes Element einer Liste (head):

$$\lambda xs.\texttt{case}_{\texttt{List}}\ xs\ \texttt{of}\ \{\texttt{Nil} \to \bot; (\texttt{Cons}\ y\ ys) \to y\}$$

Restliste ohne erstes Element (tail):

$$\lambda xs.\texttt{case}_{\texttt{List}}\ xs\ \texttt{of}\ \{\texttt{Nil} \to \bot; (\texttt{Cons}\ y\ ys) \to ys\}$$

$\bot$ repräsentiert Fehler, z.B Ω

Test, ob Liste leer ist (null):

$$\lambda xs.\texttt{case}_{\texttt{List}}\ xs\ \texttt{of}\ \{\texttt{Nil} \to \texttt{True}; (\texttt{Cons}\ y\ ys) \to \texttt{False}\}$$

If-Then-Else:

$\texttt{if}\ e\ \texttt{then}\ s\ \texttt{else}\ t$:

$$\texttt{case}_{\texttt{Bool}}\ e\ \texttt{of}\ \{\texttt{True} \to s; \texttt{False} \to t\}$$

Beispiele (2)

- Paare: Typ `Paar` mit zweistelligem Konstruktor `Paar`
  Z.B. wird $(\texttt{True}, \texttt{False})$ durch $(\texttt{Paar True False})$ dargestellt.
- Projektionen:

$$\begin{array}{lcl} \mathit{fst} & := & \lambda x.\texttt{case}_{\texttt{Paar}}\ x\ \texttt{of}\ \{(\texttt{Paar}\ a\ b) \to a\} \\ \mathit{snd} & := & \lambda x.\texttt{case}_{\texttt{Paar}}\ x\ \texttt{of}\ \{(\texttt{Paar}\ a\ b) \to b\} \end{array}$$

- Analog: mehrstellige Tupel
  Schreibweise $(a_1, \ldots, a_n)$

Haskell vs. KFPT: case-Ausdrücke (1)

Vergleich mit Haskells `case`-Ausdrücken

- Syntax ähnlich:
  - Statt $\rightarrow$ in Haskell: `->`
  - Keine Typmarkierung am `case`

  Beispiel:
  - KFPT: $\texttt{case}_{List}\ xs\ \texttt{of}\ \{\texttt{Nil} \rightarrow \texttt{Nil}; (\texttt{Cons}\ y\ ys) \rightarrow y\}$
  - Haskell: `case xs of {[] -> []; (y:ys) -> y}`

Haskell vs. KFPT: case-Ausdrücke (1)

Vergleich mit Haskells `case`-Ausdrücken

- Syntax ähnlich:
  - Statt $\rightarrow$ in Haskell: `->`
  - Keine Typmarkierung am `case`

  Beispiel:
  - KFPT: $\texttt{case}_{List}\ xs\ \texttt{of}\ \{\texttt{Nil} \rightarrow \texttt{Nil}; (\texttt{Cons}\ y\ ys) \rightarrow y\}$
  - Haskell: `case xs of {[] -> []; (y:ys) -> y}`
- In Haskell ist es nicht notwendig alle Konstruktoren abzudecken
- Kann Laufzeitfehler geben:

```
(case True of False -> False)
*** Exception: Non-exhaustive patterns in case
```

Haskell vs. KFPT: case-Ausdrücke (2)

- Haskell erlaubt überlappende Pattern und geschachtelte Pattern. Z.B. ist

  `case [] of {[] -> []; (x:(y:ys)) -> [y]; x -> []}`

  ein gültiger Haskell-Ausdruck
- Semikolon und Klammern kann man bei Einrückung weglassen:

```
case [] of
          [] -> []
          (x:(y:ys)) -> [y]
          x -> []
```

Haskell vs. KFPT: case-Ausdrücke (3)

Übersetzung von geschachtelten in einfache Pattern (für KFPT)

```
case [] of {[] -> []; (x:(y:ys)) -> [y]}
```

wird übersetzt in:

$$\begin{aligned}
&\texttt{case}_{\texttt{List}}\ \texttt{Nil of}\ \{\texttt{Nil} \to \texttt{Nil};\\
&\qquad (\texttt{Cons x z}) \to \texttt{case}_{\texttt{List}}\ \texttt{z of}\ \{\texttt{Nil} \to \bot;\\
&\qquad\qquad\qquad (\texttt{Cons y ys}) \to (\texttt{Cons y Nil})\\
&\qquad\qquad\qquad \}\\
&\qquad \}
\end{aligned}$$

- Fehlende Alternativen werden durch $Pat \to \bot$ ergänzt.
- $\bot$ (gesprochen „bot“): Repräsentant eines geschlossenen nicht terminierenden Ausdrucks.
- Abkürzung: $(\texttt{case}_{Typ}\ s\ \texttt{of}\ Alts)$

Freie und gebundene Variablen in KFPT

Zusätzlich zum Lambda-Kalkül: In case-Alternative

$$(c_i\ x_1\ \ldots\ x_{ar(c_i)}) \to s$$

sind die Variablen $x_1, \ldots, x_{ar(c_i)}$ in s gebunden.

Freie Variablen

$$
\begin{aligned}
&FV(x) && = x \\
&FV(\lambda x.s) && = FV(s) \setminus \{x\} \\
&FV(s\ t) && = FV(s) \cup FV(t) \\
&FV(c\ s_1\ \ldots\ s_{ar(c)}) && = FV(s_1) \cup \ldots \cup FV(s_{ar(c_i)}) \\
&FV(\texttt{case}_{Typ}\ t\ \texttt{of} && = FV(t) \cup (\bigcup_{i=1}^{n} (FV(s_i) \setminus \{x_{i,1}, \ldots, x_{i,ar(c_i)}\})) \\
&\quad \{(c_1\ x_{1,1}\ \ldots\ x_{1,ar(c_1)}) \to s_1; \\
&\quad \ldots \\
&\quad (c_n\ x_{n,1}\ \ldots\ x_{n,ar(c_n)}) \to s_n\})
\end{aligned}
$$

Gebundene Variablen

$$
\begin{array}{ll}
BV(x) & = \emptyset \\
BV(\lambda x.s) & = BV(s) \cup \{x\} \\
BV(s\ t) & = BV(s) \cup BV(t) \\
BV(c\ s_1\ \ldots\ s_{ar(c)}) & = BV(s_1) \cup \ldots \cup BV(s_{ar(c_i)}) \\
BV(\texttt{case}_{Typ}\ t\ \texttt{of} & = BV(t) \cup (\bigcup_{i=1}^{n} (BV(s_i) \cup \{x_{i,1}, \ldots, x_{i,ar(c_i)}\})) \\
\quad \{(c_1\ x_{1,1}\ \ldots\ x_{1,ar(c_1)}) \to s_1; & \\
\quad \ldots & \\
\quad (c_n\ x_{n,1}\ \ldots\ x_{n,ar(c_n)}) \to s_n\}) &
\end{array}
$$

Beispiel

$$s := ((\lambda x.\texttt{case}_{\texttt{List}}\ x\ \texttt{of}\ \{\texttt{Nil} \to x; \texttt{Cons}\ x\ xs\ \to \lambda u.(x\ \lambda x.(x\ u))\})\ x)$$

$FV(s) = \{x\}$ und $BV(s) = \{x, xs, u\}$

Alpha-äquivalenter Ausdruck:

$$s' := ((\lambda x_1.\texttt{case}_{\texttt{List}}\ x_1\ \texttt{of}\ \{\texttt{Nil} \to x_1; \texttt{Cons}\ x_2\ xs\ \to \lambda u.(x_2\ \lambda x_3.(x_3\ u))\})\ x)$$

$FV(s') = \{x\}$ und $BV(s') = \{x_1, x_2, xs, x_3, u\}$

Notationen

Wie im Lambda-Kalkül (mit angepasster FV, BV Definition)

- Offene und geschlossene Ausdrücke
- α-Umbenennung
- Distinct Variable Convention

Operationale Semantik

Substitution

- $s[t/x]$ ersetzt alle freien Vorkommen von x in s durch t
  (wenn $BV(s) \cap FV(t) = \emptyset$)
- $s[t_1/x_1, \ldots, t_n/x_n]$ parallele Ersetzung von $x_1, \ldots, x_n$ durch $t_1, \ldots, t_n$
  (wenn für alle i:$BV(s) \cap FV(t_i) = \emptyset$)

Operationale Semantik

Substitution

- $s[t/x]$ ersetzt alle freien Vorkommen von x in s durch t
  (wenn $BV(s) \cap FV(t) = \emptyset$)
- $s[t_1/x_1, \ldots, t_n/x_n]$ parallele Ersetzung von $x_1, \ldots, x_n$ durch $t_1, \ldots, t_n$
  (wenn für alle i:$BV(s) \cap FV(t_i) = \emptyset$)

Definition

Die Reduktionsregeln (β) und (case) sind in KFPT definiert als:

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

$$\text{(case)} \quad \texttt{case}_{Typ}\ (c\ s_1\ \ldots\ s_{ar(c)})\ \texttt{of}\ \{\ldots;\ (c\ x_1 \ldots\ x_{ar(c)}) \to t;\ \ldots\} \to t[s_1/x_1, \ldots, s_{ar(c)}/x_{ar(c)}]$$

Wenn $r_1 \to r_2$ mit (β) oder (case) dann reduziert r_1 unmittelbar zu r_2

Beispiel

$$
\begin{array}{ll}
 & (\lambda x.\texttt{case}_{\texttt{Paar}}\ x\ \texttt{of}\ \{(\texttt{Paar}\ a\ b) \to a\})\ (\texttt{Paar True False}) \\
\xrightarrow{\beta} & \texttt{case}_{\texttt{Paar}}\ (\texttt{Paar True False})\ \texttt{of}\ \{(\texttt{Paar}\ a\ b) \to a\} \\
\xrightarrow{case} & \texttt{True}
\end{array}
$$

KFPT-Kontexte

Kontext = Ausdruck mit Loch $[\cdot]$

$$
\begin{aligned}
\mathbf{Ctxt} ::= {} & [\cdot] \mid \lambda V.\mathbf{Ctxt} \mid (\mathbf{Ctxt}\ \mathbf{Expr}) \mid (\mathbf{Expr}\ \mathbf{Ctxt}) \\
\mid {} & (c_i\ \mathbf{Expr}_1\ \ldots \mathbf{Expr}_{i-1}\ \mathbf{Ctxt}\ \mathbf{Expr}_{i+1}\ \mathbf{Expr}_{ar(c_i)}) \\
\mid {} & (\mathtt{case}_{\mathtt{Typ}}\ \mathbf{Ctxt}\ \mathtt{of}\ \{\mathbf{Pat}_1 \to \mathbf{Expr}_1; \ldots; \mathbf{Pat}_n \to \mathbf{Expr}_n\}) \\
\mid {} & (\mathtt{case}_{\mathtt{Typ}}\ \mathbf{Expr}\ \mathtt{of}\ \{\mathbf{Pat}_1 \to \mathbf{Expr}_1; \ldots; \mathbf{Pat}_i \to \mathbf{Ctxt}; \ldots, \mathbf{Pat}_n \to \mathbf{Expr}_n\})
\end{aligned}
$$

Wenn $C[s] \to C[t]$ wobei $s \xrightarrow{\beta} t$ oder $s \xrightarrow{case} t$, dann bezeichnet man s (mit seiner Position in C) als Redex von $C[s]$.

Redex = Reducible expression

Call-by-Name-Reduktion

Definition

Reduktionskontexte R in KFPT werden durch die folgende Grammatik mit Startsymbol $\mathbf{RCtxt}$ erzeugt:

$$\mathbf{RCtxt} ::= [\cdot] \mid (\mathbf{RCtxt}\ \mathbf{Expr}) \mid (\texttt{case}_{Typ}\ \mathbf{RCtxt}\ \texttt{of}\ Alts)$$

Call-by-Name-Reduktion (2)

Definition

Wenn r_1 unmittelbar zu r_2 reduziert, dann ist $R[r_1] \to R[r_2]$ für jeden Reduktionskontext R eine Call-by-Name-Reduktion.

- Notation: $\xrightarrow{name}$, bzw. auch $\xrightarrow{name,\beta}$ und $\xrightarrow{name,case}$.
- $\xrightarrow{name,+}$ transitive Hülle von $\xrightarrow{name}$
- $\xrightarrow{name,*}$ reflexiv-transitive Hülle von $\xrightarrow{name}$

Beispiele

$$\begin{array}{lcl}(\lambda x.x)\ ((\lambda y.y)\ (\lambda z.z)) & \to & x[(\lambda y.y)\ (\lambda z.z)/x] \\ & = & (\lambda y.y)\ (\lambda z.z)\end{array}$$

ist eine Call-by-Name-Reduktion.

$$\begin{array}{lcl}(\lambda x.x)\ ((\lambda y.y)\ (\lambda z.z)) & \to & (\lambda x.x)\ (y[(\lambda z.z)/y]) \\ & = & (\lambda x.x)\ (\lambda z.z)\end{array}$$

ist keine Call-by-Name-Reduktion

Normalformen

Ein KFPT-Ausdruck s ist eine

- Normalform (NF = normal form):
  s enthält keine (β)- oder (case)-Redexe und ist WHNF
- Kopfnormalform (HNF = head normal form):
  s ist Konstruktoranwendung oder Abstraktion $\lambda x_1, \ldots x_n.s'$, wobei s' entweder Variable oder $(c\ s_1\ \ldots\ s_{ar(c)})$ oder $(x\ s')$ ist
- schwache Kopfnormalform (WHNF = weak head normal form):
  s ist eine FWHNF oder eine CWHNF.
- funktionale schwache Kopfnormalform (FWHNF = functional whnf):
  s ist eine Abstraktion
- Konstruktor-schwache Kopfnormalform (CWHNF = constructor whnf):
  s ist eine Konstruktoranwendung $(c\ s_1\ \ldots\ s_{ar(c)})$

Wir verwenden nur WHNFs (keine NFs, keine HNFs).

Terminierung bzw. Konvergenz

Definition

Ein KFPT-Ausdruck s konvergiert (oder *terminiert*, notiert als $s{\downarrow}$) genau dann, wenn:

$$s{\downarrow} \iff \exists \text{ WHNF } t : s \xrightarrow{name,*} t$$

Falls s nicht konvergiert, so sagen wir s divergiert und notieren dies mit $s{\Uparrow}$.

Sprechweisen:

Wir sagen s hat eine WHNF (bzw. FWHNF, CWHNF),
wenn s zu einer WHNF (bzw. FWHNF, CWHNF) mit $\xrightarrow{name,*}$
reduziert werden kann.

Dynamische Typisierung

Call-by-Name-Reduktion stoppt ohne WHNF: Folgende Fälle:

- eine freie Variable ist potentieller Redex
  (Ausdruck von der Form $R[x]$), oder
- ein dynamischer Typfehler tritt auf.

Dynamische Typisierung

Call-by-Name-Reduktion stoppt ohne WHNF: Folgende Fälle:

- eine freie Variable ist potentieller Redex
  (Ausdruck von der Form $R[x]$), oder
- ein dynamischer Typfehler tritt auf.

Definition (Dynamische Typregeln für KFPT)

Ein KFPT-Ausdruck s direkt dynamisch ungetypt, falls:

- $s = R[\texttt{case}_T\ (c\ s_1\ \ldots\ s_n)\ \texttt{of}\ Alts]$ und c ist *nicht* vom Typ T
- $s = R[\texttt{case}_T\ \lambda x.t\ \texttt{of}\ Alts]$.
- $s = R[(c\ s_1\ \ldots\ s_{ar(c)})\ t]$

Dynamische Typisierung

Call-by-Name-Reduktion stoppt ohne WHNF: Folgende Fälle:

- eine freie Variable ist potentieller Redex
  (Ausdruck von der Form $R[x]$), oder
- ein dynamischer Typfehler tritt auf.

Definition (Dynamische Typregeln für KFPT)

Ein KFPT-Ausdruck s direkt dynamisch ungetypt, falls:

- $s = R[\mathtt{case}_T\ (c\ s_1\ \ldots\ s_n)\ \mathtt{of}\ Alts]$ und c ist *nicht* vom Typ T
- $s = R[\mathtt{case}_T\ \lambda x.t\ \mathtt{of}\ Alts]$.
- $s = R[(c\ s_1\ \ldots\ s_{ar(c)})\ t]$

$$s \text{ ist dynamisch ungetypt}$$
$$\Longleftrightarrow$$
$$\exists t : s \xrightarrow{name,*} t \wedge t \text{ ist direkt dynamisch ungetypt}$$

Beispiele

- $\texttt{case}_{\texttt{List}}\ \texttt{True}\ \texttt{of}\ \{\texttt{Nil} \to \texttt{Nil}; (\texttt{Cons}\ x\ xs) \to xs\}$
  ist direkt dynamisch ungetypt
- $(\lambda x.\texttt{case}_{\texttt{List}}\ x\ \texttt{of}\ \{\texttt{Nil} \to \texttt{Nil}; (\texttt{Cons}\ x\ xs) \to xs\})\ \texttt{True}$
  ist dynamisch ungetypt
- $(\texttt{Cons}\ \texttt{True}\ \texttt{Nil})\ (\lambda x.x)$ ist direkt dynamisch ungetypt
- $(\texttt{case}_{\texttt{Bool}}\ x\ \texttt{of}\ \{\texttt{True} \to \texttt{True}; \texttt{False} \to \texttt{False}\})$
  ist nicht (direkt) dynamisch ungetypt
- $(\lambda x.\texttt{case}_{\texttt{Bool}}\ x\ \texttt{of}\ \{\texttt{True} \to \texttt{True}; \texttt{False} \to \texttt{False}\})\ (\lambda y.y)$
  ist dynamisch ungetypt

Dynamische Typisierung (2)

Satz

Ein geschlossener KFPT-Ausdruck s ist irreduzibel (bzgl. der Call-by-Name-Auswertung) genau dann, wenn eine der folgenden Bedingungen auf ihn zutrifft:

- Entweder ist s eine WHNF, oder
- s ist direkt dynamisch ungetypt.

Die folgende Eigenschaft wird auch Progress-Lemma genannt:

> Wenn t geschlossen, keine WHNF und getypt,
> dann kann man eine Call-by-Name-Reduktion durchführen.

Markierungsalgorithmus zur Call-by-Name-Redex-Suche

Für Ausdruck s starte mit $s^\star$, Verschieberegeln:

- $(s\ t)^\star \Rightarrow (s^\star\ t)$
- $(\texttt{case}_{Typ}\ s\ \texttt{of}\ Alts)^\star \Rightarrow (\texttt{case}_{Typ}\ s^\star\ \texttt{of}\ Alts)$

Markierungsalgorithmus zur Call-by-Name-Redex-Suche

Für Ausdruck s starte mit $s^\star$, Verschieberegeln:

- $(s\ t)^\star \Rightarrow (s^\star\ t)$
- $(\texttt{case}_{Typ}\ s\ \texttt{of}\ Alts)^\star \Rightarrow (\texttt{case}_{Typ}\ s^\star\ \texttt{of}\ Alts)$

Fälle danach:

- Markierung ist an einer Abstraktion; Fälle:
  - $(\lambda x.s')^\star$, dann FWHNF
  - $C[((\lambda x.s')^\star\ s'')]$, dann reduziere die Applikation mit β
  - $C[\texttt{case}_T\ (\lambda x.s')^\star\ \ldots)]$, dann direkt dynamisch ungetypt

Markierungsalgorithmus zur Call-by-Name-Redex-Suche

Für Ausdruck s starte mit $s^\star$, Verschieberegeln:

- $(s\ t)^\star \Rightarrow (s^\star\ t)$
- $(\texttt{case}_{Typ}\ s\ \texttt{of}\ Alts)^\star \Rightarrow (\texttt{case}_{Typ}\ s^\star\ \texttt{of}\ Alts)$

Fälle danach:

- Markierung ist an einer Abstraktion; Fälle:
  - $(\lambda x.s')^\star$, dann FWHNF
  - $C[((\lambda x.s')^\star\ s'')]$, dann reduziere die Applikation mit β
  - $C[\texttt{case}_T\ (\lambda x.s')^\star\ \ldots)]$, dann direkt dynamisch ungetypt
- Markierung ist an einer Konstruktorapplikation
  - $(c\ \ldots)^\star$, dann CWHNF
  - $C[((c\ \ldots)^\star\ s')]$, dann direkt dynamisch ungetypt
  - $C[(\texttt{case}_T\ (c\ \ldots)^\star\ alts)]$, reduzieren, falls c zum Typ T gehört, sonst direkt dynamisch ungetypt

Markierungsalgorithmus zur Call-by-Name-Redex-Suche

Für Ausdruck s starte mit $s^\star$, Verschieberegeln:

- $(s\ t)^\star \Rightarrow (s^\star\ t)$
- $(\texttt{case}_{Typ}\ s\ \texttt{of}\ Alts)^\star \Rightarrow (\texttt{case}_{Typ}\ s^\star\ \texttt{of}\ Alts)$

Fälle danach:

- Markierung ist an einer Abstraktion; Fälle:
  - $(\lambda x.s')^\star$, dann FWHNF
  - $C[((\lambda x.s')^\star\ s'')]$, dann reduziere die Applikation mit β
  - $C[\texttt{case}_T\ (\lambda x.s')^\star\ \ldots)]$, dann direkt dynamisch ungetypt
- Markierung ist an einer Konstruktorapplikation
  - $(c\ \ldots)^\star$, dann CWHNF
  - $C[((c\ \ldots)^\star\ s')]$, dann direkt dynamisch ungetypt
  - $C[(\texttt{case}_T\ (c\ \ldots)^\star\ alts)]$, reduzieren, falls c zum Typ T gehört, sonst direkt dynamisch ungetypt
- Markierung ist an einer Variablen: Keine Reduktion möglich

Beispiel

$$(((\lambda x.\lambda y.(\begin{pmatrix}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to (x\ z)\}\end{pmatrix}\ \texttt{True}))\ (\lambda u, v.v))\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))^*$$

Beispiel

$$(((\lambda x.\lambda y.(\begin{pmatrix}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to (x\ z)\}\end{pmatrix}\ \texttt{True}))\ (\lambda u, v.v))^{\star}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

Beispiel

$$(((\lambda x.\lambda y.(\begin{pmatrix}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to (x\ z)\}\end{pmatrix}\ \texttt{True}))^{\star}\ (\lambda u, v.v))\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

Beispiel

$$(((\lambda x.\lambda y.(\begin{pmatrix}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to (x\ z)\}\end{pmatrix}\ \texttt{True}))^{\star}\ (\lambda u, v.v))\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

$$\xrightarrow{name,\beta} \quad ((\lambda y.(\begin{pmatrix}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{pmatrix}\ \texttt{True}))\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))^{\star}$$

Beispiel

$$(((\lambda x.\lambda y.(\begin{pmatrix}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to (x\ z)\}\end{pmatrix}\ \texttt{True}))^{\star}\ (\lambda u, v.v))\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

$$\xrightarrow{name,\beta} \quad ((\lambda y.(\begin{pmatrix}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{pmatrix}\ \texttt{True}))^{\star}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

Beispiel

$$(((\lambda x.\lambda y.(\begin{pmatrix}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to (x\ z)\}\end{pmatrix}\ \texttt{True}))^{\star}\ (\lambda u, v.v))\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

$$\xrightarrow{name,\beta} \quad ((\lambda y.(\begin{pmatrix}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{pmatrix}\ \texttt{True}))^{\star}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

$$\xrightarrow{name,\beta} \quad (\begin{pmatrix}\texttt{case}_{\texttt{List}}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil})\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{pmatrix}\ \texttt{True})^{\star}$$

Beispiel

$$(((\lambda x.\lambda y.(\begin{pmatrix}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to (x\ z)\}\end{pmatrix}\ \texttt{True}))^{\star}\ (\lambda u, v.v))\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

$$\xrightarrow{name,\beta} \quad ((\lambda y.(\begin{pmatrix}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{pmatrix}\ \texttt{True}))^{\star}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

$$\xrightarrow{name,\beta} \quad (\begin{pmatrix}\texttt{case}_{\texttt{List}}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil})\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{pmatrix}^{\star}\ \texttt{True})$$

Beispiel

$$(((\lambda x.\lambda y.(\begin{pmatrix}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to (x\ z)\}\end{pmatrix}\ \texttt{True}))^{\star}\ (\lambda u, v.v))\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

$$\xrightarrow{name,\beta} \quad ((\lambda y.(\begin{pmatrix}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{pmatrix}\ \texttt{True}))^{\star}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

$$\xrightarrow{name,\beta} \quad (\begin{pmatrix}\texttt{case}_{\texttt{List}}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil})^{\star}\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{pmatrix}\ \texttt{True})$$

Beispiel

$$(((\lambda x.\lambda y.(\begin{pmatrix}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to (x\ z)\}\end{pmatrix}\ \texttt{True}))^{\star}\ (\lambda u, v.v))\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

$$\xrightarrow{name,\beta} \quad ((\lambda y.(\begin{pmatrix}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{pmatrix}\ \texttt{True}))^{\star}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

$$\xrightarrow{name,\beta} \quad (\begin{pmatrix}\texttt{case}_{\texttt{List}}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil})^{\star}\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{pmatrix}\ \texttt{True})$$

$$\xrightarrow{name,case} (((\lambda u, v.v)\ (\lambda w.w))\ \texttt{True})^{\star}$$

Beispiel

$$(((\lambda x.\lambda y.(\begin{pmatrix}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to (x\ z)\}\end{pmatrix}\ \texttt{True}))^{\star}\ (\lambda u, v.v))\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

$$\xrightarrow{name,\beta} \quad ((\lambda y.(\begin{pmatrix}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{pmatrix}\ \texttt{True}))^{\star}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

$$\xrightarrow{name,\beta} \quad (\begin{pmatrix}\texttt{case}_{\texttt{List}}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil})^{\star}\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{pmatrix}\ \texttt{True})$$

$$\xrightarrow{name,case} (((\lambda u, v.v)\ (\lambda w.w))^{\star}\ \texttt{True})$$

Beispiel

$$(((\lambda x.\lambda y.(\begin{pmatrix}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to (x\ z)\}\end{pmatrix}\ \texttt{True}))^{\star}\ (\lambda u, v.v))\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

$$\xrightarrow{name,\beta} \quad ((\lambda y.(\begin{pmatrix}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{pmatrix}\ \texttt{True}))^{\star}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

$$\xrightarrow{name,\beta} \quad (\begin{pmatrix}\texttt{case}_{\texttt{List}}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil})^{\star}\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{pmatrix}\ \texttt{True})$$

$$\xrightarrow{name,case} (((\lambda u, v.v)^{\star}\ (\lambda w.w))\ \texttt{True})$$

Beispiel

$$(((\lambda x.\lambda y.(\begin{pmatrix}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to (x\ z)\}\end{pmatrix}\ \texttt{True}))^{\star}\ (\lambda u, v.v))\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

$$\xrightarrow{name,\beta} \quad ((\lambda y.(\begin{pmatrix}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{pmatrix}\ \texttt{True}))^{\star}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

$$\xrightarrow{name,\beta} \quad (\begin{pmatrix}\texttt{case}_{\texttt{List}}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil})^{\star}\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{pmatrix}\ \texttt{True})$$

$$\xrightarrow{name,case} (((\lambda u, v.v)^{\star}\ (\lambda w.w))\ \texttt{True})$$

$$\xrightarrow{name,\beta} \quad ((\lambda v.v)\ \texttt{True})^{\star}$$

Beispiel

$$(((\lambda x.\lambda y.(\begin{pmatrix}\mathtt{case}_{\mathtt{List}}\ y\ \mathtt{of}\ \{\\ \mathtt{Nil} \to \mathtt{Nil};\\ (\mathtt{Cons}\ z\ zs) \to (x\ z)\}\end{pmatrix}\ \mathtt{True}))^{\star}\ (\lambda u, v.v))\ (\mathtt{Cons}\ (\lambda w.w)\ \mathtt{Nil}))$$

$$\xrightarrow{name,\beta} \quad ((\lambda y.(\begin{pmatrix}\mathtt{case}_{\mathtt{List}}\ y\ \mathtt{of}\ \{\\ \mathtt{Nil} \to \mathtt{Nil};\\ (\mathtt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{pmatrix}\ \mathtt{True}))^{\star}\ (\mathtt{Cons}\ (\lambda w.w)\ \mathtt{Nil}))$$

$$\xrightarrow{name,\beta} \quad (\begin{pmatrix}\mathtt{case}_{\mathtt{List}}\ (\mathtt{Cons}\ (\lambda w.w)\ \mathtt{Nil})^{\star}\ \mathtt{of}\ \{\\ \mathtt{Nil} \to \mathtt{Nil};\\ (\mathtt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{pmatrix}\ \mathtt{True})$$

$$\xrightarrow{name,case} (((\lambda u, v.v)^{\star}\ (\lambda w.w))\ \mathtt{True})$$

$$\xrightarrow{name,\beta} \quad ((\lambda v.v)^{\star}\ \mathtt{True})$$

Beispiel

$$(((\lambda x.\lambda y.(\begin{pmatrix}\mathtt{case}_{\mathtt{List}}\ y\ \mathtt{of}\ \{\\ \mathtt{Nil} \to \mathtt{Nil};\\ (\mathtt{Cons}\ z\ zs) \to (x\ z)\}\end{pmatrix}\ \mathtt{True}))^{\star}\ (\lambda u, v.v))\ (\mathtt{Cons}\ (\lambda w.w)\ \mathtt{Nil}))$$

$$\xrightarrow{name,\beta} \quad ((\lambda y.(\begin{pmatrix}\mathtt{case}_{\mathtt{List}}\ y\ \mathtt{of}\ \{\\ \mathtt{Nil} \to \mathtt{Nil};\\ (\mathtt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{pmatrix}\ \mathtt{True}))^{\star}\ (\mathtt{Cons}\ (\lambda w.w)\ \mathtt{Nil}))$$

$$\xrightarrow{name,\beta} \quad (\begin{pmatrix}\mathtt{case}_{\mathtt{List}}\ (\mathtt{Cons}\ (\lambda w.w)\ \mathtt{Nil})^{\star}\ \mathtt{of}\ \{\\ \mathtt{Nil} \to \mathtt{Nil};\\ (\mathtt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{pmatrix}\ \mathtt{True})$$

$$\xrightarrow{name,case} (((\lambda u, v.v)^{\star}\ (\lambda w.w))\ \mathtt{True})$$

$$\xrightarrow{name,\beta} \quad ((\lambda v.v)^{\star}\ \mathtt{True})$$

$$\xrightarrow{name,\beta} \quad \mathtt{True}$$

Darstellung von Ausdrücken als Termgraphen

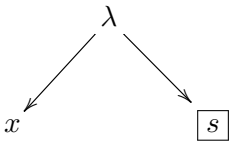
Knoten für je ein syntaktisches Konstrukt des Ausdrucks

- Variablen = ein Blatt

Darstellung von Ausdrücken als Termgraphen

Knoten für je ein syntaktisches Konstrukt des Ausdrucks

- Variablen = ein Blatt
- Abstraktionen $\lambda x.s$

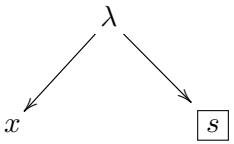


wobei $\boxed{s}$ Baum für s

Darstellung von Ausdrücken als Termgraphen

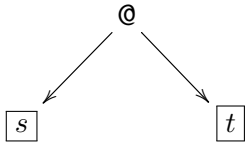
Knoten für je ein syntaktisches Konstrukt des Ausdrucks

- Variablen = ein Blatt
- Abstraktionen $\lambda x.s$



wobei $\boxed{s}$ Baum für s

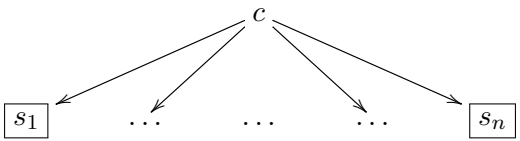
- Applikationen $(s\ t)$



wobei $\boxed{s}$, $\boxed{t}$ Bäume für s und t

Darstellung von Ausdrücken als Termgraphen (2)

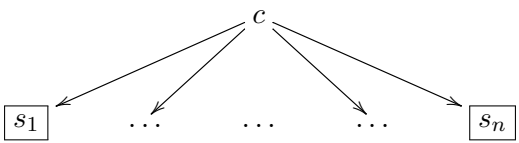
- Konstruktoranwendungen n-stellig: $(c\ s_1 \ldots\ s_n)$



wobei $\boxed{s_i}$ die Bäume für s_i

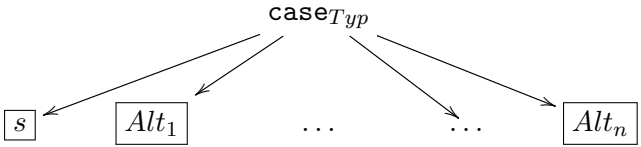
Darstellung von Ausdrücken als Termgraphen (2)

- Konstruktoranwendungen n-stellig: $(c\ s_1 \ldots\ s_n)$



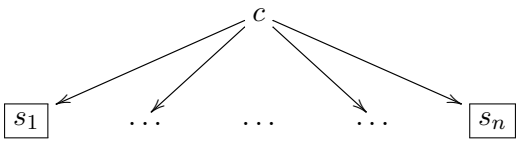
wobei $\boxed{s_i}$ die Bäume für s_i

- case-Ausdrücke: $n+1$ Kinder, $\texttt{case}_{Typ}\ s\ \texttt{of}\ \{Alt_1; \ldots; Alt_n\}$



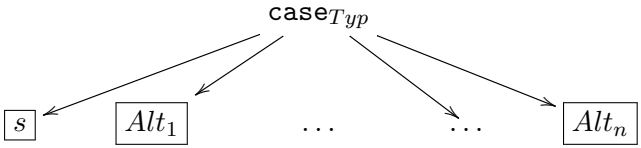
Darstellung von Ausdrücken als Termgraphen (2)

- Konstruktoranwendungen n-stellig: $(c\ s_1 \ldots\ s_n)$

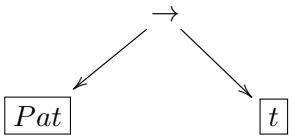


wobei $\boxed{s_i}$ die Bäume für s_i

- case-Ausdrücke: $n+1$ Kinder, $\texttt{case}_{Typ}\ s\ \texttt{of}\ \{Alt_1; \ldots; Alt_n\}$



- case-Alternative $Pat \to t$

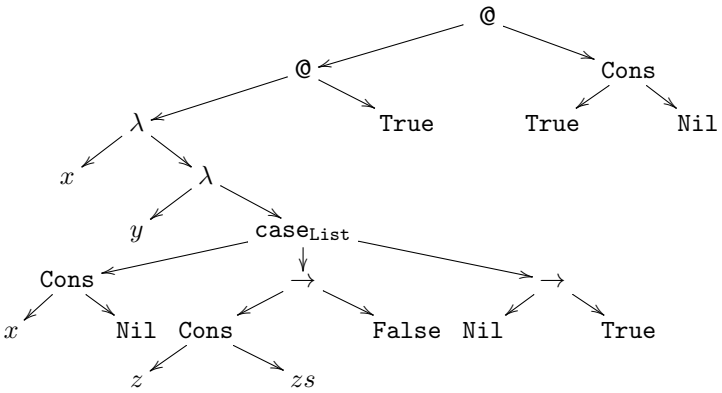


Beispiel

$$\left(\left(\begin{array}{l}\lambda x.\lambda y.\mathtt{case}_{\mathtt{List}}\ (\mathtt{Cons}\ x\ \mathtt{Nil})\ \mathtt{of}\ \{\\ \qquad (\mathtt{Cons}\ z\ zs) \to \mathtt{False};\\ \qquad \mathtt{Nil} \to \mathtt{True}\}\end{array}\right)\ \mathtt{True}\right)\ (\mathtt{Cons}\ \mathtt{True}\ \mathtt{Nil})$$

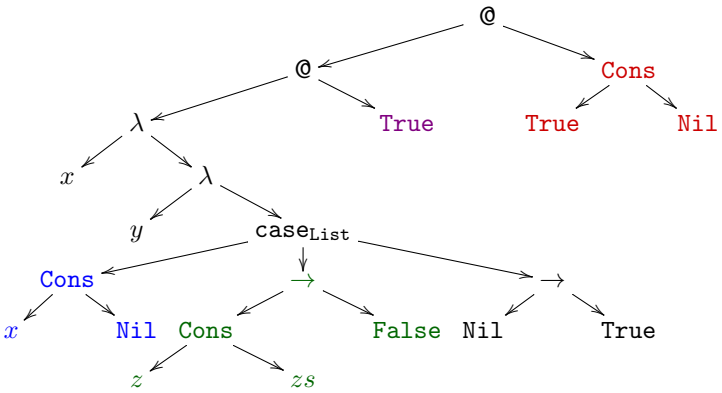
Beispiel

$$\left(\left(\begin{array}{l}\lambda x.\lambda y.\texttt{case}_{\texttt{List}}\ (\texttt{Cons}\ x\ \texttt{Nil})\ \texttt{of}\ \{\\ \qquad (\texttt{Cons}\ z\ zs) \to \texttt{False};\\ \qquad \texttt{Nil} \to \texttt{True}\}\end{array}\right)\ \texttt{True}\right)\ (\texttt{Cons}\ \texttt{True}\ \texttt{Nil})$$



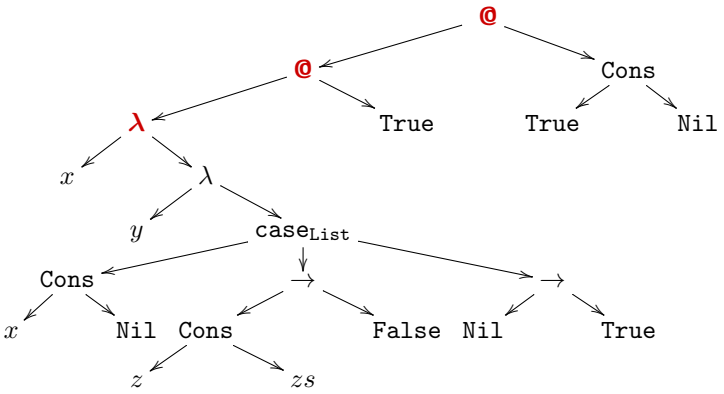
Beispiel

$$\left(\left(\begin{array}{l}\lambda x.\lambda y.\texttt{case}_{\texttt{List}}\ (\texttt{Cons}\ x\ \texttt{Nil})\ \texttt{of}\ \{\\ \quad (\texttt{Cons}\ z\ zs) \to \texttt{False};\\ \quad \texttt{Nil} \to \texttt{True}\}\end{array}\right)\ \texttt{True}\right)\ (\texttt{Cons}\ \texttt{True}\ \texttt{Nil})$$



Beispiel

$$\left(\left(\begin{array}{l}\lambda x.\lambda y.\texttt{case}_{\texttt{List}}\ (\texttt{Cons}\ x\ \texttt{Nil})\ \texttt{of}\ \{\\ \qquad (\texttt{Cons}\ z\ zs) \to \texttt{False};\\ \qquad \texttt{Nil} \to \texttt{True}\}\end{array}\right)\ \texttt{True}\right)\ (\texttt{Cons}\ \texttt{True}\ \texttt{Nil})$$



CBN-Redex-Suche: immer links, bis Abstraktion, Konstruktoranwendung

Call-by-Name-Reduktion: Eigenschaften

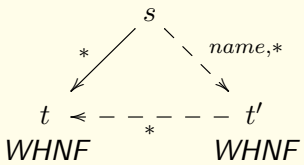
- Die Call-by-Name-Reduktion ist deterministisch, d.h. für jedes s gibt es höchstens ein t mit $s \xrightarrow{name} t$.
- Eine WHNF ist irreduzibel bezüglich der Call-by-Name-Reduktion.

Call-by-Name-Reduktion: Eigenschaften

- Die Call-by-Name-Reduktion ist deterministisch, d.h. für jedes s gibt es höchstens ein t mit $s \xrightarrow{name} t$.
- Eine WHNF ist irreduzibel bezüglich der Call-by-Name-Reduktion.

Theorem (Standardisierung für KFPT)

Wenn $s \xrightarrow{*} t$ mit beliebigen (β)- und (case)-Reduktionen (in beliebigem Kontext angewendet), wobei t eine WHNF ist, dann existiert eine WHNF t', so dass $s \xrightarrow{name,*} t'$ und $t' \xrightarrow{*} t$ (unter α-Gleichheit).



Die Kernsprache KFPTS

Rekursive Superkombinatoren: KFPTS

- **Nächste Erweiterung:** KFPT zu KFPTS
- „S“ steht für Superkombinatoren
- Superkombinatoren sind Namen (Konstanten) für Funktionen
- Superkombinatoren dürfen auch rekursive Funktionen sein

Annahme: Es gibt eine Menge von Superkombinatornamen $\mathcal{SK}$.

Beispiel: Superkombinator `length`

$$\begin{aligned}\texttt{length}\ xs = &\texttt{case}_{\texttt{List}}\ xs\ \texttt{of}\ \{\\ &\texttt{Nil} \to 0;\\ &(\texttt{Cons}\ y\ ys) \to (1 + \texttt{length}\ ys)\}\end{aligned}$$

KFPTS: Syntax

$$
\begin{aligned}
\mathbf{Expr} ::= {} & V \mid \lambda V.\mathbf{Expr} \mid (\mathbf{Expr}_1\ \mathbf{Expr}_2) \\
\mid {} & (c_i\ \mathbf{Expr}_1\ \ldots\ \mathbf{Expr}_{ar(c_i)}) \\
\mid {} & (\mathtt{case}_{\mathtt{Typ}}\ \mathbf{Expr}\ \mathtt{of}\ \{\mathbf{Pat}_1 \to \mathbf{Expr}_1; \ldots; \mathbf{Pat}_n \to \mathbf{Expr}_n\}) \\
\mid {} & SK \text{ wobei } SK \in \mathcal{SK}
\end{aligned}
$$

$\mathbf{Pat}_i$ ::= $(c_i\ V_1 \ldots V_{ar(c_i)})$ wobei die Variablen V_i alle verschieden sind.

KFPTS: Syntax (2)

Zu jedem Superkombinator SK gibt es eine **Superkombinatordefinition**:

$$SK\ V_1\ \ldots\ V_n = \mathbf{Expr}$$

dabei

- V_i paarweise verschiedene Variablen
- $\mathbf{Expr}$ ein KFPTS-Ausdruck
- $FV(\mathbf{Expr}) \subseteq \{V_1, \ldots, V_n\}$
- $ar(SK) = n \geq 0$: Stelligkeit des Superkombinators ($n = 0$ ist möglich).

Beispiel: Definition des Superkombinators map:

$$map\ f\ xs = \texttt{case}_{\texttt{List}}\ xs\ \texttt{of}\ \{\texttt{Nil} \to \texttt{Nil}; (\texttt{Cons}\ y\ ys) \to \texttt{Cons}\ (f\ y)\ (map\ f\ ys)\}$$

KFPTS: Syntax (3)

Ein KFPTS-Programm besteht aus:

- Einer Menge von Typen und Konstruktoren
- einer Menge von Superkombinator-Definitionen
- einem KFPTS-Ausdruck s

Dabei müssen alle in s verwendeten Superkombinatoren auch definiert sein.

KFPTS:Syntax (2) Superkombinatoren

Haskell: Unterschiede bei Superkombinatordefinitionen:

- Mehrere Definitionen (für verschiedene Fälle) pro Superkombinator
- Argumente können Pattern sein
- Guards sind möglich

KFPTS: Operationale Semantik

Reduktionskontexte:

$$\mathbf{RCtxt} ::= [\cdot] \mid (\mathbf{RCtxt}\ \mathbf{Expr}) \mid \texttt{case}_{Typ}\ \mathbf{RCtxt}\ \texttt{of}\ Alts$$

KFPTS: Operationale Semantik

Reduktionskontexte:

$$\mathbf{RCtxt} ::= [\cdot] \mid (\mathbf{RCtxt}\ \mathbf{Expr}) \mid \texttt{case}_{Typ}\ \mathbf{RCtxt}\ \texttt{of}\ Alts$$

Reduktionsregeln (β), (case) und (SK-β):

(β) $\quad (\lambda x.s)\ t \to s[t/x]$

(case) $\quad \texttt{case}_{Typ}\ (c\ s_1\ \ldots\ s_{ar(c)})\ \texttt{of}\ \{\ldots;\ (c\ x_1 \ldots\ x_{ar(c)}) \to t;\ \ldots\}$
$\to t[s_1/x_1, \ldots, s_{ar(c)}/x_{ar(c)}]$

(SK-β) $\quad (SK\ s_1\ \ldots\ s_n) \to e[s_1/x_1, \ldots, s_n/x_n]$,
wenn $SK\ x_1\ \ldots\ x_n = e$ die Definition von SK ist

KFPTS: Operationale Semantik

Reduktionskontexte:

$$\mathbf{RCtxt} ::= [\cdot] \mid (\mathbf{RCtxt}\ \mathbf{Expr}) \mid \texttt{case}_{Typ}\ \mathbf{RCtxt}\ \texttt{of}\ Alts$$

Reduktionsregeln (β), (case) und (SK-β):

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

$$\text{(case)} \qquad \texttt{case}_{Typ}\ (c\ s_1\ \ldots\ s_{ar(c)})\ \texttt{of}\ \{\ldots;\ (c\ x_1 \ldots\ x_{ar(c)}) \to t;\ \ldots\} \to t[s_1/x_1, \ldots, s_{ar(c)}/x_{ar(c)}]$$

$$\text{(SK-}\beta\text{)} \qquad (SK\ s_1\ \ldots\ s_n) \to e[s_1/x_1, \ldots, s_n/x_n], \text{ wenn } SK\ x_1\ \ldots\ x_n = e \text{ die Definition von } SK \text{ ist}$$

Call-by-Name-Reduktion:

$$\frac{s \to t \text{ mit } (\beta)\text{-, (case)- oder (SK-}\beta\text{)}}{R[s] \xrightarrow{name} R[t]}$$

KFPTS: WHNFs und Dynamische Typisierung

WHNFs

- WHNF = CWHNF oder FWHNF
- CWHNF = Konstruktoranwendung $(c\ s_1\ \ldots\ s_{ar(c)})$
- FWHNF = Abstraktion oder $SK\ s_1\ \ldots\ s_m$ mit $ar(SK) > m$

Direkt dynamisch ungetypt:

- Regeln wie vorher in KFPT: $R[(\texttt{case}_T\ \lambda x.s\ \texttt{of}\ \ldots)]$, $R[(\texttt{case}_T\ (c\ s_1 \ldots s_n)\ \texttt{of}\ \ldots)]$, wenn c nicht von Typ T und $R[((c\ s_1\ \ldots\ s_{ar(c)})\ t)]$
- Neue Regel:
  $R[\texttt{case}_T\ (SK\ s_1\ \ldots\ s_m)\ \texttt{of}\ Alts]$ ist direkt dynamisch ungetypt, falls $ar(SK) > m$.

Markierungsalgorithmus

Markierung funktioniert genauso wie in KFPTS:

- $(s\ t)^\star \Rightarrow (s^\star\ t)$
- $(\texttt{case}_{Typ}\ s\ \texttt{of}\ Alts)^\star \Rightarrow (\texttt{case}_{Typ}\ s^\star\ \texttt{of}\ Alts)$

Neue Fälle:

- Ein Superkombinator ist mit $\star$ markiert:
  - Genügend Argumente vorhanden: Reduziere mit (SK-β)
  - Zu wenig Argumente und kein Kontext außen: WHNF
  - Zu wenig Argumente und im Kontext (case [.] ...): direkt dynamisch ungetypt

Beispiel

Die Superkombinatoren map und not seien definiert als:

$$
\begin{aligned}
map\ f\ xs &= \texttt{case}_{\texttt{List}}\ xs\ \texttt{of}\ \{\texttt{Nil} \to \texttt{Nil}; \\
&\qquad\qquad\qquad\qquad (\texttt{Cons}\ y\ ys) \to \texttt{Cons}\ (f\ y)\ (map\ f\ ys)\} \\
not\ x &= \texttt{case}_{\texttt{Bool}}\ x\ \texttt{of}\ \{\texttt{True} \to \texttt{False}; \texttt{False} \to \texttt{True}\}
\end{aligned}
$$

Beispiel

Die Superkombinatoren map und not seien definiert als:

$$\begin{aligned} map\ f\ xs &= \texttt{case}_{\texttt{List}}\ xs\ \texttt{of}\ \{\texttt{Nil} \to \texttt{Nil}; \\ &\qquad\qquad (\texttt{Cons}\ y\ ys) \to \texttt{Cons}\ (f\ y)\ (map\ f\ ys)\} \\ not\ x &= \texttt{case}_{\texttt{Bool}}\ x\ \texttt{of}\ \{\texttt{True} \to \texttt{False}; \texttt{False} \to \texttt{True}\} \end{aligned}$$

Beispiel zur Auswertung:

$$\begin{aligned} & map\ not\ (\texttt{Cons True}\ (\texttt{Cons False Nil})) \\ \xrightarrow{name,\text{SK-}\beta} \quad & \texttt{case}_{\texttt{List}}\ (\texttt{Cons True}\ (\texttt{Cons False Nil}))\ \texttt{of}\ \{ \\ & \quad \texttt{Nil} \to \texttt{Nil}; \\ & \quad (\texttt{Cons}\ y\ ys) \to \texttt{Cons}\ (not\ y)\ (map\ not\ ys)\} \\ \xrightarrow{name,\text{case}} \quad & \texttt{Cons}\ (not\ \texttt{True})\ (map\ not\ (\texttt{Cons False Nil})) \end{aligned}$$

WHNF erreicht!

Beachte: Im GHCI-Interpreter wird nur aufgrund des Anzeigens weiter ausgewertet

Erweiterung um `seq`

Erweiterung um seq

seq ist in KFPT, KFPTS nicht kodierbar!

Wir bezeichnen mit

- KFPT+seq die Erweiterung von KFPT um seq
- KFPTS+seq die Erweiterung von KFPTS um seq

Wir verzichten auf die formale Definition!
Man benötigt u.a. die Reduktionsregel:

$$\mathsf{seq}\ v\ t \quad \to \quad t, \text{ wenn } v \text{ WHNF}$$

erweiterte Reduktionskontexte und die neue Verschieberegel:

$$(\mathsf{seq}\ s\ t)^\star \quad \to \quad (\mathsf{seq}\ s^\star\ t)$$

Erweiterung um polymorphe Typen: KFPTSP

KFPTSP

Mit KFPTSP bezeichnen wir polymorph getyptes KFPTS

Definition

Die Syntax von polymorphen Typen kann durch die folgende Grammatik beschrieben werden:

$$\mathbf{T} ::= TV \mid TC\ \mathbf{T}_1\ \ldots\ \mathbf{T}_n \mid \mathbf{T}_1 \to \mathbf{T}_2$$

wobei TV für eine Typvariable steht und TC ein Typkonstruktor mit Stelligkeit n ist.

Beispiele

$$
\begin{array}{lcl}
\texttt{True} & :: & \texttt{Bool} \\
\texttt{False} & :: & \texttt{Bool} \\
\texttt{not} & :: & \texttt{Bool} \to \texttt{Bool} \\
\texttt{map} & :: & (a \to b) \to [a] \to [b] \\
(\lambda x.x) & :: & (a \to a)
\end{array}
$$

Einfache Typregeln

- Für die Anwendung:

$$\frac{s :: T_1 \to T_2, t :: T_1}{(s\ t) :: T_2}$$

- Instantiierung

$$\frac{s :: T}{s :: T'}$$ wenn $T' = \sigma(T)$, wobei σ eine Typsubstitution ist, die Typen für Typvariablen ersetzt.

- Für case-Ausdrücke:

$$\frac{s :: T_1, \quad \forall i : Pat_i :: T_1, \quad \forall i : t_i :: T_2}{(\texttt{case}_T\ s\ \texttt{of}\ \{Pat_1 \to t_1; \ldots; Pat_n \to t_n) :: T_2\}}$$

Beispiel

$$\begin{aligned}\texttt{and} &:= \lambda x, y.\texttt{case}_{\texttt{Bool}}\ x\ \texttt{of}\ \{\texttt{True} \to y; \texttt{False} \to \texttt{False}\}\\ \texttt{or} &:= \lambda x.y.\texttt{case}_{\texttt{Bool}}\ x\ \texttt{of}\ \{\texttt{True} \to \texttt{True}; \texttt{False} \to y\}\end{aligned}$$

Mit der Anwendungsregel:

$$\frac{\dfrac{\texttt{and} :: \texttt{Bool} \to \texttt{Bool} \to \texttt{Bool}, \texttt{True} :: \texttt{Bool}}{(\texttt{and True}) :: \texttt{Bool} \to \texttt{Bool}}, \texttt{False} :: \texttt{Bool}}{(\texttt{and True False}) :: \texttt{Bool}}$$

Beispiel

$$\frac{\texttt{True} :: \texttt{Bool},\ \texttt{False} :: \texttt{Bool}' \qquad \dfrac{\dfrac{\dfrac{\texttt{Cons} :: a \to [a] \to [a]}{\texttt{Cons} :: \texttt{Bool} \to [\texttt{Bool}] \to [\texttt{Bool}]},\ \texttt{True} :: \texttt{Bool}}{(\texttt{Cons}\ \texttt{True}) :: [\texttt{Bool}] \to [\texttt{Bool}]},\ \dfrac{\texttt{Nil} :: [a]}{\texttt{Nil} :: [\texttt{Bool}]}}{(\texttt{Cons}\ \texttt{True}\ \texttt{Nil}) :: [\texttt{Bool}]},\ \frac{\texttt{Nil} :: [a]}{\texttt{Nil} :: [\texttt{Bool}]}}{\texttt{case}_{\texttt{Bool}}\ \texttt{True}\ \texttt{of}\ \{\texttt{True} \to (\texttt{Cons}\ \texttt{True}\ \texttt{Nil}); \texttt{False} \to \texttt{Nil}\} :: [\texttt{Bool}]}$$

Beispiel

$$\dfrac{\dfrac{\texttt{map} :: (a \to b) \to [a] \to [b]}{\texttt{map} :: (\texttt{Bool} \to \texttt{Bool}) \to [\texttt{Bool}] \to [\texttt{Bool}]}, \texttt{not} :: \texttt{Bool} \to \texttt{Bool}}{(\texttt{map not}) :: [\texttt{Bool}] \to [\texttt{Bool}]}$$

KFPTSP+seq als Kernsprache für Haskell

- Haskell hat weitere syntaktische Konstrukte, die KFPTSP+seq nicht hat.
- Wir argumentieren, wie diese verlustfrei in KFPTSP+seq übersetzt werden können.
- Wir betrachten dabei nur den funktionalen Teil von Haskell

let und where

- Nicht-rekursives `let` lässt sich übersetzen:
  `let` $x = s$ `in` $t \rightsquigarrow (\lambda x.t)\ s$.
- Rekursive `let`-Ausdrücke: Komplizierter (z.B. durch Verwendung von Fixpunktkombinatoren)
- where-Klauseln gehen analog

Zahlen

- Endlicher Wertebereich: Als Typ mit (vielen) Konstruktoren
- Unendlicher Wertebereich: Z.B. als Liste von Bits
- Andere Möglichkeit: Peano-Zahlen

```
data Pint = Zero | Succ Pint
 deriving(Eq,Show)
```

$$\begin{aligned} \mathcal{P}(0) &:= \texttt{Zero} \\ \mathcal{P}(n) &:= \texttt{Succ}(\mathcal{P}(n-1)) \text{ für } n > 0 \end{aligned}$$

Z.B. 3 als $\texttt{Succ}(\texttt{Succ}(\texttt{Succ}(\texttt{Zero})))$.

Funktionen, für Peano-Zahlen

```
-- Test auf gültige Zahl
istZahl :: Pint -> Bool
istZahl x =  case x of {Zero  -> True; (Succ y) ->  istZahl y}
-- beachte:
unendlich :: Pint
unendlich = Succ unendlich
-- Addition
peanoPlus :: Pint -> Pint -> Pint
peanoPlus x y = if istZahl x &&  istZahl y then plus x y else bot
  where plus x y = case x of
                     Zero   -> y
                     Succ z -> Succ (plus z y)
-- Multiplikation
peanoMult :: Pint -> Pint -> Pint
peanoMult x y  = if istZahl x &&  istZahl y then mult x y else bot
  where mult x y = case x of
                     Zero   -> Zero
                     Succ z -> peanoPlus y (mult z y)
```

Funktionen, für Peano-Zahlen (2)

```
peanoEq :: Pint -> Pint -> Bool
peanoEq x y = if istZahl x &&  istZahl y then eq x y else bot
 where eq Zero Zero          = True
       eq (Succ x) (Succ y)  = eq x y
       eq _        _         = False

peanoLeq :: Pint -> Pint -> Bool
peanoLeq x y = if istZahl x &&  istZahl y then leq x y else bot
 where leq Zero y            = True
       leq x    Zero         = False
       leq (Succ x) (Succ y) = leq x y
```

if-then-else und case

- if-then-else-Ausdrücke werden in case-Ausdrücke übersetzt
- case-Ausdrücke in Haskell bieten mehr Möglichkeiten:
  - Verschachtelte Pattern
  - Default-Alternativen
  - Fehlende Pattern
- Übersetzung durch:
  - Verschachtelte Pattern werden in verschachtelte case-Ausdrücke mit einfachen Pattern übersetzt.
  - Fehlende Pattern werden hinzugefügt, wobei die entsprechenden case-Alternativen auf einen nichttermineren Ausdruck $\perp$ abgebildet werden.
  - Default-Alternativen werden durch alle nicht abgedeckten Pattern ersetzt.

case-Ausdrücke: Beispiel

```
case ys of
  (x1:(x2:(x3:(x4:xs)))) -> Just x4
  _ ->  Nothing
```

wird zu (noch in Haskell-Syntax):

```
case ys of
  [] -> Nothing
  (x1:ys') -> case ys' of
                [] -> Nothing
                (x2:ys'') -> case ys'' of
                               [] -> Nothing
                               (x3:ys''') -> case ys''' of
                                               [] -> Nothing
                                               (x4:xs) -> Just x4
```

Funktionsdefinitionen

- Mehrzeilige Definition mit Pattern und Guards werden in eine Superkombinatordefinition und case-Ausdrücke ersetzt.
- Pattern werden zu case-Ausdrücken
- Guards werden zu if-then-else (und dann zu case)
- Datenfluss beachten

Funktionen: Beispiele

```
f (x:xs)
  | x > 10  = True
  | x < 100 = True
f ys        = False
```

wird zu:

```
f ys = case ys of {
        Nil -> False;
        (x:xs) -> if x > 10 then True else
                    if x < 100 then True else False
       }
```

Funktionen: Beispiele (2)

```
myFun frucht p personendb =
  case frucht of
    Banane l h kruemmung
      | Just x <- lookup p personendb -> Right x
    _  -> Left "Fehler"
```

wird zu:

```
myFun fruct p personendb =
   case frucht of
     Banane l h kruemmung -> case lookup p personendb of
                               Just x -> Right x
                               Nothing -> Left "Fehler"
     Apfel x1 -> Left "Fehler"
     Birne x2 x3 -> Left "Fehler"
```

Übersicht

Kernsprache	**Besonderheiten**
KFPT	Erweiterung des call-by-name Lambda-Kalküls um (schwach) getyptes case und Datenkonstruktoren, seq ist nicht kodierbar.
KFPTS	Erweiterung von KFPT um rekursive Superkombinatoren, seq nicht kodierbar.
KFPTSP	KFPTS, polymorph getypt; seq nicht kodierbar.
KFPT+seq	Erweiterung von KFPT um den seq-Operator
KFPTS+seq	Erweiterung von KFPTS um den seq-Operator
KFPTSP+seq	KFPTSP+seq mit polymorpher Typisierung, geeignete Kernsprache für Haskell

Lazy Evaluation in Haskell

Lazy Evaluation in Haskell

- Wir führen keine KFPTSP+seq Variante mit Call-by-Need-Auswertung ein
  Grund: Zu kompliziert
- Stattdessen beschreiben wir aus praktischer Sicht, wie der GHC effizient (mit Sharing) auswertet.

Ideen:

- Ausdrücke werden als Graphen repräsentiert
- Ein unausgewerteter (Unter-)Ausdruck ist dann ein Verweis auf einen Teilgraphen und wird als Thunk bezeichnet.
- Bei der ersten Verwendung des Verweises wird der Ausdruck ausgewertet und durch sein Ergebniswert ersetzt, weitere Verwendungen des Verweises liefern sofort den Ergebniswert

Beispiel

Betrachte `square (10+5)` wobei

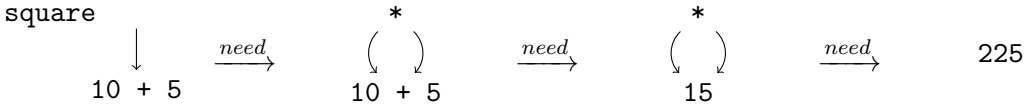
```
square x = x*x
```

- Call-by-Name-Auswertung:
  `square (10+5)` $\xrightarrow{name}$ `(10+5)*(10+5)` $\xrightarrow{name}$ `15*(10+5)` $\xrightarrow{name}$ `15*15` $\xrightarrow{name}$ 225.
- Call-by-Value-Auswertung:
  `square (10+5)` $\xrightarrow{value}$ `square 15` $\xrightarrow{value}$ `15*15` $\xrightarrow{value}$ 225.
- Call-by-Need-Auswertung ($\langle i \rangle$) repräsentieren Referenzen

  (`square` $\langle 1 \rangle$, $\{\langle 1 \rangle \mapsto$ `10+5`$\}$)
  $\xrightarrow{need}$ ($\langle 1 \rangle$ * $\langle 1 \rangle$, $\{\langle 1 \rangle \mapsto$ `10+5`$\}$)
  $\xrightarrow{need}$ (15 * 15, $\{\langle 1 \rangle \mapsto 15\}$)
  $\xrightarrow{need}$ (225, $\{\langle 1 \rangle \mapsto 15\}$).

Beispiel: Als Darstellung mit Graphen

Eine alternative Darstellung ist die Verwendung von Graphen:



Beispiel mit Trace

```
import Debug.Trace          -- von Verwendung wird abgeraten!!
-- trace :: String -> a -> a  -- Seiteneffekt: Textausgabe

foo :: Int -> Int -> Int -> Int
foo x y z = y + y + z

z = foo (trace "first"  1)
        (trace "second" 2)
        (trace "third"  3)
```

Auswertung von `z`

- Call-by-Value: `"first" "second" "third" 7`
- Call-by-Name: `"second" "second" "third" 7`
- Call-by-Need: `"second" "third" 7` .

Beispiel mit Divergenz

```
foo x y z  = if x < 0 then abs x else x+y
```

Auswertung einer Anwendung `foo` e_1 e_2 e_3:

- e_1, e_2 und e_3 werden nicht angefasst, sondern in den Graphen (Heap) eingefügt und `x`, `y` und `z` sind Verweise auf die Ausdrücke e_1, e_2 und e_3.
- Im Anschluss beginnt die Auswertung des Rumpfs von `foo`.
- Die Auswertung des `if`-Ausdrucks erfordert ein Auswerten von `x<0` und dieses wiederum ein Auswerten des Arguments `x`.
- Daher wird e_1 ausgewertet und durch das Resultat ersetzt.
- Falls `x<0` gilt, wird der Wert von `abs x` zurückgegeben; weder e_2 noch e_3 werden ausgewertet.
- Falls `x<0` falsch ist, wird der Wert von `x+y` zurückgegeben; dies erfordert die Auswertung von e_2. Daher wird e_3 in keinem Fall ausgewertet.

Der Ausdruck `foo 1 2 (1 `div` 0)` ist daher wohldefiniert.

Noch ein Beispiel

```
g x y =   let z = x `div` y
          in if x > y * y then x else z
t2 = g 5 0
```

- Wertet man `t2` aus, so erhält man `5`.
- Da der Wert von `z` nicht benötigt wird, kommt es nicht zur Division durch Null.
- Sequentialität der Auswertung (erst das, dann das) ist daher nicht gegeben, was manchmal verwirrend sein kann.

Potentiell unendliche Datenstrukturen

Lazy Evaluation ermöglicht „unendliche“ Datenstrukturen:

```
ones  =  1 : ones       -- "unendliche" Liste von 1en
twos  =  map (1+) ones  -- "unendliche" Liste von 2en
nums  = iterate (1+) 0  -- Liste der natürlichen Zahlen

iterate :: (a -> a) -> a -> [a]
iterate f x  =   x : iterate f (f x)

> take 10 nums
[0,1,2,3,4,5,6,7,8,9]
```

Es wird immer nur soviel von der Datenstruktur ausgewertet, wie benötigt wird:

```
nums = iterate (1+) 0

> take 10 nums
[0,1,2,3,4,5,6,7,8,9]
```

Thunks im GHCi

- Im GHCi ist es tlw. möglich, den Speicher zu inspizieren und zu beobachten, welche Teile einer Datenstruktur durch die Auswertung ausgewertet wurden und welche nicht.
- Die beiden Kommandos hierzu sind `:print` und `:sprint` (die zweite Variante liefert eine vereinfachte Ansicht im Vergleich zur ersten).
- Der Aufruf ist `:sprint [<name> ...]`. Die Speicherzustände der definierten Namen in `<name> ...` werden angezeigt, wobei unausgewertete Thunks als _ dargestellt sind.

```
Prelude> let x = map even [1..22]
Prelude> let y = (map odd [10..20],x)
Prelude> :sprint x y
x = _
y = (_,_)
Prelude> take 3 $ drop 3 x
[True,False,True]
Prelude> :sprint x y
x = _ : _ : _ : True : False : True : _
y = (_,_ : _ : _ : True : False : True : _)
```

Thunks im GHCi

Mit :print:

```
Prelude> let x = map even [1..22]
Prelude> let y = (map odd [10..20],x)
Prelude> :print x y
x = (_t1::[Bool])
y = ((_t2::[Bool]),(_t3::[Bool]))
Prelude> take 3 $ drop 3 x
[True,False,True]
Prelude> :print x y
x = (_t4::Bool) : (_t5::Bool) : (_t6::Bool) : True : False : True :
    (_t7::[Bool])
y = ((_t8::[Bool]),(_t9::Bool) : (_t10::Bool) : (_t11::Bool) :
                   True : False : True : (_t12::[Bool]))
```

Thunks im GHCi (3)

Man beachte, dass das Anzeigen nur für monomorphe Typen funktioniert, z.B. ergibt

```
Prelude> let z =  map (*2) [1..6]::[Int]
Prelude> take 3 $ drop 3 z
[8,10,12]
Prelude> :sprint z
z = _ : _ : _ : 8 : 10 : 12 : _
Prelude> let z =  map (*2) [1..6]
Prelude> take 3 $ drop 3 z
[8,10,12]
Prelude> :sprint z
z = _
Prelude> :print z
z = (_t30::(Num b, Enum b) => [b])
```

Thunks im GHCi (4)

Mit `:force` kann man die Auswertung von Thunks außerhalb der Reihenfolge erzwingen:

```
Prelude> let x = map even [1..22]
Prelude> :sprint x
x = _
Prelude> take 3 $ drop 3 x
[True,False,True]
Prelude> :print x
x = (_t1::Bool) : (_t2::Bool) : (_t3::Bool) : True : False : True :
    (_t4::[Bool])
Prelude> :force _t2
_t2 = True
Prelude> :print x
x = (_t5::Bool) : True : (_t6::Bool) : True : False : True :
    (_t7::[Bool])
```

Trennung von Daten und Kontrollfluss

```
factors :: Integral a => a -> [a]
factors n = filter (\m -> n `mod` m == 0) [2 .. (n - 1)]

isPrime :: Integral a => a -> Bool
isPrime n = n > 1 && null (factors n)
```

Die Funktion `factors` berechnet alle Faktoren einer Zahl. Die Berechnung von `isPrime` bricht sofort ab, sobald der erste Faktor gefunden wurde, trotz Verwendung von `factors` werden also nicht alle Faktoren berechnet!

Trennung von Daten und Kontrollfluss (2)

Weiteres Beispiel: Sieb des Erathostenes

```
primes :: [Integer]
primes   =   sieve [2..]

sieve :: [Integer] -> [Integer]
sieve (p:xs) = p : sieve (xs `minus` [p,p+p..])

minus xs@(x:xt) ys@(y:yt) = case compare x y of
  LT -> x : minus xt ys
  EQ ->     minus xt yt
  GT ->     minus xs yt
```

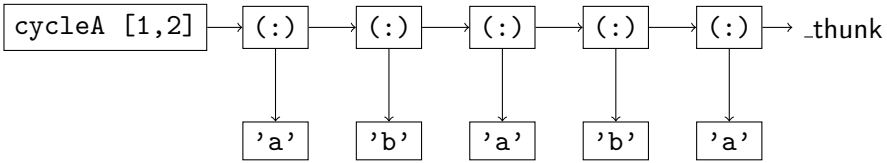
Zirkularität

```
(++) :: [a] -> [a] -> [a]
(++) []     ys = ys
(++) (x:xs) ys = x : xs ++ ys


cycleA xs = xs ++ cycleA xs

cycleB xs = xs' where xs' = xs++xs'
```

- `cycleA l` verbraucht potenziell unendlich viel Speicher, bzw. genauer so viel Speicherzellen, wie Elemente von `cycleA l` gelesen werden.
- Für eine Liste `l` mit Länge n verbraucht `cycleB` jedoch nur maximal n Speicherzellen.

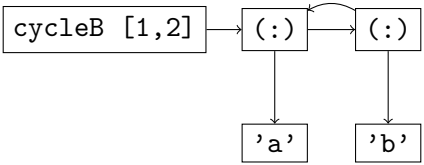
Zirkularität (2)

Call-by-Need-Auswertung von `cycleA [1,2]` nach Anforderung einiger Elemente:



Zirkularität (3)

Call-by-Need-Auswertung von `cycleB [1,2]` nach Anforderung einiger Elemente:



Strikte Auswertung

- Bereits gesehen: `seq` und `$!`-Operator
- Dabei wird nur bis zur WHNF ausgewertet
- Tiefer auswerten: `Control.DeepSeq`

Spracherweiterung `BangPatterns`:
Patterns, welche mit `!` beginnen, müssen immer zuerst ausgewertet werden:

```
stack exec -- ghci -XBangPatterns
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude> let foo (x,!y) = [x,y]
Prelude> drop 1 $ foo (undefined,2)
[2]
Prelude> take 1 $ foo (2,undefined)
*** Exception: Prelude.undefined
```

Strikte Datentypen

Der GHC erlaubt auch die Deklaration von strikten Datentypen:

```
data FaulesPaar   = FP  Integer  Bool deriving Show
data StriktesPaar = SP !Integer !Bool deriving Show
```

```
Prelude>  let (FP u v) = FP undefined True in v
True
Prelude> let (SP u v) = SP undefined True in v
*** Exception: Prelude.undefined
```

- Der Konstruktor `SP` wird hier immer mit ausgewerteten Argumenten abgespeichert;
- die Argumente von `FP` können dagegen thunks sein.

Strikte Datentypen (2)

Die Spracherweiterung `StrictData` macht alle Datentypen strikt; lazy Argumente erhält man dann mit einer Tilde:

```
stack exec -- ghci -XStrictData
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude> data Paar = P Integer Bool
Prelude> let (P a b) = P undefined True in b
*** Exception: Prelude.undefined
Prelude> data FaulesPaar = FP ~Integer ~Bool
Prelude> let (FP a b) = FP undefined True in b
True
```

Lazy Pattern

Lazy Pattern-Matches: Versehe das Pattern mit ~:

```
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude> let bar (x,y) = 42
Prelude> bar undefined
*** Exception: Prelude.undefined
Prelude> let baz ~(x,y) = 42
Prelude> baz undefined
42
```

- Lazy Patterns matchen immer (sind daher *irrefutable*)
- Diese Pattern immer nur als letzte Möglichkeit verwenden.

Falsch:

```
mylength ~[] = 0
mylength (_:xs) = 1 + mylength xs
```

- Beachte das Pattern auf linken Seiten von `let`-Ausdrücken ebenfalls stets lazy sind.

Lazy-Pattern (2)

Die Verwendung der Lazy-Pattern kann man sich so vorstellen: Aus

```
quu ~(x,y) = x+y
```

wird

```
quu p = let (x,y) = p in x+y
```

was wiederum übersetzt wird in

```
quu p = let x = case p of (x,_) -> x
            y = case p of (_,y) -> y
        in (x+y)
```

oder kürzer geschrieben:

```
quu p = (fst p) +(snd p)
```

Beispiel

Funktion `splitAt`:

```
splitAt :: Int -> [a] -> ([a], [a])
splitAt n ls
  | n <= 0       = ([], ls)
splitAt _ []     = ([], [])
splitAt n (y:ys) =
    case splitAt (n-1) ys of
      ~(prefix, suffix) -> (y : prefix, suffix)
```

Durch den Lazy-Pattern-Match steht der erste Teil des Ergebnis-Paares sofort zur Verfügung. Z.B. liefert

```
head $ fst $ splitAt 10000000 (repeat 'a')
```

sofort `'a'` zurück.
Bei Verwendung eines normalen Pattern Matches, dauert dies viel länger
(da erst 10 Millionen rekursive Aufrufe berechnet werden)