GOETHE UNIVERSITÄT FRANKFURT AM MAIN

# Einführung in die Funktionale Programmierung:

## Funktionale Kernsprachen:

## Die KFP-Kernsprachen

Prof. Dr. Manfred Schmidt-Schauß

WS 2022/23

Stand der Folien: 16. November 2022

# Übersicht

# Lambda-Kalkül und Haskell

Berechnung im Lambda-Kalkül ist universell.

# Lambda-Kalkül und Haskell

Berechnung im Lambda-Kalkül ist universell.

ABER:

- Es ist sehr mühsam etwas zu programmieren
- Es ist ebenfalls mühsam aus der Ausgabe eine verständliche Antwort zu extrahieren.
- Sehr wenig intuitiv
- Kein automatischer Standard für z.B. Zahlen und Listen usw.
- Iteration und Rekursion nur über Fixpunkt-Kombinatoren

# Lambda-Kalkül und Haskell

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

# Lambda-Kalkül und Haskell

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

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

# Lambda-Kalkül und Haskell

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

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

  Ausweg mittels **Church-Kodierung**:

  z.B. *true* $= \lambda x, y.x$
  *false* $= \lambda x, y.y$
  *if-then-else* $= \lambda b, x_1, x_2.b\ x_1\ x_2$

  *if-then-else true* $e_1\ e_2 = (\lambda b, x_1, x_2.b\ x_1\ x_2)\ (\lambda x, y.x)\ e_1\ e_2$
  $\xrightarrow{no,\beta,3} (\lambda x, y.x)\ e_1\ e_2 \xrightarrow{no,\beta,2} e_1$

  *if-then-else false* $e_1\ e_2 = (\lambda b, x_1, x_2.b\ x_1\ x_2)\ (\lambda x, y.y)\ e_1\ e_2$
  $\xrightarrow{no,\beta,3} (\lambda x, y.y)\ e_1\ e_2 \xrightarrow{no,\beta,2} e_2$

  Aber: Kompliziert; Daten und Funktionen sind nicht unterscheidbar; Typisierung passt nicht zur Intuition und zu Haskell; Was sind gleiche Ergebnisse?

# Lambda-Kalkül und Haskell (2)

**Im Lambda Kalkül:**

- Rekursive Funktionen: Geht nur über Fixpunkt-Kombinatoren:

  Fakultät als Beispiel:

$$
\begin{aligned}
fak = {} & (\lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))) \\
& (\lambda f.\lambda x.\mathtt{if}\ x = 0\ \mathtt{then}\ 1\ \mathtt{else}\ x * (f\ (x-1)))
\end{aligned}
$$

  Aber: Kompliziert und schwer verständlich!

# Lambda-Kalkül und Haskell (2)

## Im Lambda Kalkül:

- Rekursive Funktionen: Geht nur über Fixpunkt-Kombinatoren:

  Fakultät als Beispiel:

$$\begin{aligned}fak = (\lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x)))\\ (\lambda f.\lambda x.\mathtt{if}\ x = 0\ \mathtt{then}\ 1\ \mathtt{else}\ x * (f\ (x-1)))\end{aligned}$$

  Aber: Kompliziert und schwer verständlich!

- Typisierung fehlt! Aber Haskell ist polymorph getypt.

# Lambda-Kalkül und Haskell (2)

**Im Lambda Kalkül:**

- Rekursive Funktionen: Geht nur über Fixpunkt-Kombinatoren:

  Fakultät als Beispiel:

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

  Aber: Kompliziert und schwer verständlich!
- Typisierung fehlt! Aber Haskell ist polymorph getypt.
- Kein `seq`: In Haskell ist `seq` verfügbar, im Lambda-Kalkül nicht kodierbar.

# Kernsprachen für Haskell

- Im folgenden führen wir eine Hierarchie von Kernsprachen ein.
  $\lambda$-Kalkül $\subset$ KFPT ... $\subset$ ... Haskell
  ($\subset$ für Konstrukte, nicht die vollen Sprachen)
- Alle sind Erweiterungen des Lambda-Kalküls
- Bezeichnungen: KFP...
- KFP = Kern einer Funktionalen Programmiersprache

# 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

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

**Annahmen:**

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

**Beispiele**

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

# Datentypen

**Annahmen:**

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

**Beispiele**

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

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

  Beachte $[a_1, a_2, \ldots, a_n]$ ist Abkürzung für $a_1 : (a_2 : (\ldots : (a_n : [])))$

# Syntax von KFPT

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

# Syntax von KFPT

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

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

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

Restliste ohne erstes Element (tail):

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

$\bot$ repräsentiert Fehler, z.B $\Omega$

# Beispiele (1)

Erstes Element einer Liste (head):

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

Restliste ohne erstes Element (tail):

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

$\bot$ repräsentiert Fehler, z.B $\Omega$

Test, ob Liste leer ist (null):

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

# Beispiele (1)

Erstes Element einer Liste (head):

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

Restliste ohne erstes Element (tail):

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

$\bot$ repräsentiert Fehler, z.B $\Omega$

Test, ob Liste leer ist (null):

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

If-Then-Else:

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

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

# Beispiele (2)

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

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

- Analog: mehrstellige Tupel
- In Haskell sind Tupel bereits vorhanden (eingebaut), Schreibweise $(a_1, \ldots, a_n)$
- In Haskell auch 0-stellige Tupel, keine 1-stelligen Tupel

# 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)

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

- KFPT erlaubt nur disjunkte Pattern!
- 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 (Haskell) in einfache Pattern (für KFPT)

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

wird übersetzt in:

$$
\begin{array}{l}
\mathtt{case}_{\mathtt{List}}\ \mathtt{Nil}\ \mathtt{of}\ \{\mathtt{Nil} \to \mathtt{Nil}; \\
\qquad (\mathtt{Cons}\ \mathtt{x}\ \mathtt{z}) \to \mathtt{case}_{\mathtt{List}}\ \mathtt{z}\ \mathtt{of}\ \{\mathtt{Nil} \to \bot; \\
\qquad\qquad (\mathtt{Cons}\ \mathtt{y}\ \mathtt{ys}) \to (\mathtt{Cons}\ \mathtt{y}\ \mathtt{Nil}) \\
\qquad\qquad \} \\
\qquad \}
\end{array}
$$

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

# Freie und gebundene Variablen in KFPT

Zusätzlich zum Lambda-Kalkül:

In einer case-Alternative

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

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

# Freie Variablen

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

# Gebundene Variablen

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

# 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 in KFPT

Notationen in KFPT:
Wie im Lambda-Kalkül (mit angepasster $FV$, $BV$ Definition)

- Offene und geschlossene Ausdrücke
- $\alpha$-Umbenennung
- Distinct Variable Convention

# Operationale Semantik von KFPT

## 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 von KFPT

## 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 $(\beta)$ und (case) sind in KFPT definiert als:

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

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

Wenn $s \to t$ mit $(\beta)$ oder (case) dann reduziert $s$ unmittelbar zu $t$

# Beispiel

$$(\lambda x.\texttt{case}_{\texttt{Paar}}\ x\ \texttt{of}\ \{(\texttt{Paar}\ a\ b) \to a\})\ (\texttt{Paar True False})$$

# Beispiel

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

# 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 $[\cdot]$

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

# KFPT-Kontexte

Kontext = Ausdruck mit Loch $[\cdot]$

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

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

# Normalordnungsreduktion

## Definition

Reduktionskontexte $R$ in KFPT werden durch die folgende Grammatik erzeugt:

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

Normalordnungsreduktionen finden im Reduktionskontext statt

# Normalordnungsreduktion

## Definition

Reduktionskontexte $R$ in KFPT werden durch die folgende Grammatik erzeugt:

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

Normalordnungsreduktionen finden im Reduktionskontext statt

## Redexsuche mit $*$ zum Verschieben

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

Neue Regel:

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

# Beispiel

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

# Beispiel

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

# 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

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

$$\xrightarrow{no,\beta}\quad ((\lambda y.(\begin{pmatrix}\texttt{case}_{\texttt{List}}\ y\ \mathsf{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{pmatrix}\ \texttt{True}))\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))^{\star}$$

# Beispiel

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

$$\xrightarrow{no,\beta} \quad ((\lambda y.(\begin{pmatrix}\mathtt{case}_{\mathtt{List}}\ y\ \mathsf{of}\ \{\\ \mathtt{Nil} \to \mathtt{Nil};\\ (\mathtt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{pmatrix}\ \mathtt{True}))^{\star}\ (\mathtt{Cons}\ (\lambda w.w)\ \mathtt{Nil}))$$

# Beispiel

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

$$\xrightarrow{no,\beta}\quad ((\lambda y.(\begin{pmatrix}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{pmatrix}\ \texttt{True}))^{\star}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

$$\xrightarrow{no,\beta}\quad (\begin{pmatrix}\texttt{case}_{\texttt{List}}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil})\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{pmatrix}\ \texttt{True})^{\star}$$

# Beispiel

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

$$\xrightarrow{no,\beta}\quad ((\lambda y.\left(\begin{array}{l}\texttt{case}_{\texttt{List}}\ y\ \mathsf{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{array}\right)\ \texttt{True}))^{\star}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

$$\xrightarrow{no,\beta}\quad (\left(\begin{array}{l}\texttt{case}_{\texttt{List}}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil})\ \mathsf{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{array}\right)^{\star}\ \texttt{True})$$

# Beispiel

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

$$\xrightarrow{no,\beta} \quad ((\lambda y.(\begin{pmatrix}\texttt{case}_{\texttt{List}}\; y \;\texttt{of}\; \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\; z\; zs) \to ((\lambda u, v.v)\; z)\}\end{pmatrix}\; \texttt{True}))^{\star}\; (\texttt{Cons}\; (\lambda w.w)\; \texttt{Nil}))$$

$$\xrightarrow{no,\beta} \quad (\begin{pmatrix}\texttt{case}_{\texttt{List}}\; (\texttt{Cons}\; (\lambda w.w)\; \texttt{Nil})^{\star} \;\texttt{of}\; \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\; z\; zs) \to ((\lambda u, v.v)\; z)\}\end{pmatrix}\; \texttt{True})$$

# Beispiel

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

$$\xrightarrow{no,\beta}\quad ((\lambda y.\left(\begin{array}{l}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{array}\right)\ \texttt{True}))^{\star}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

$$\xrightarrow{no,\beta}\quad (\left(\begin{array}{l}\texttt{case}_{\texttt{List}}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil})^{\star}\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{array}\right)\ \texttt{True})$$

$$\xrightarrow{no,case}\ (((\lambda u, v.v)\ (\lambda w.w))\ \texttt{True})^{\star}$$

# Beispiel

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

$$\xrightarrow{no,\beta} \quad ((\lambda y.\left(\begin{array}{l}\texttt{case}_{\texttt{List}}\; y \;\texttt{of}\; \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\; z\; zs) \to ((\lambda u, v.v)\; z)\}\end{array}\right) \texttt{True}))^{\star}\; (\texttt{Cons}\; (\lambda w.w)\; \texttt{Nil}))$$

$$\xrightarrow{no,\beta} \quad (\left(\begin{array}{l}\texttt{case}_{\texttt{List}}\; (\texttt{Cons}\; (\lambda w.w)\; \texttt{Nil})^{\star} \;\texttt{of}\; \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\; z\; zs) \to ((\lambda u, v.v)\; z)\}\end{array}\right) \texttt{True})$$

$$\xrightarrow{no,case} (((\lambda u, v.v)\; (\lambda w.w))^{\star}\; \texttt{True})$$

# Beispiel

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

$$\xrightarrow{no,\beta} \quad ((\lambda y.\left(\begin{array}{l}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{array}\right)\ \texttt{True}))^{\star}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

$$\xrightarrow{no,\beta} \quad (\left(\begin{array}{l}\texttt{case}_{\texttt{List}}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil})^{\star}\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{array}\right)\ \texttt{True})$$

$$\xrightarrow{no,case} (((\lambda u, v.v)^{\star}\ (\lambda w.w))\ \texttt{True})$$

# Beispiel

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

$$\xrightarrow{no,\beta}\quad ((\lambda y.\left(\begin{array}{l}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{array}\right)\ \texttt{True}))^{\star}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

$$\xrightarrow{no,\beta}\quad (\left(\begin{array}{l}\texttt{case}_{\texttt{List}}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil})^{\star}\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{array}\right)\ \texttt{True})$$

$$\xrightarrow{no,case}\ (((\lambda u, v.v)^{\star}\ (\lambda w.w))\ \texttt{True})$$

$$\xrightarrow{no,\beta}\quad ((\lambda v.v)\ \texttt{True})^{\star}$$

# Beispiel

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

$$\xrightarrow{no,\beta}\quad ((\lambda y.\left(\begin{array}{l}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{array}\right)\ \texttt{True}))^{\star}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

$$\xrightarrow{no,\beta}\quad (\left(\begin{array}{l}\texttt{case}_{\texttt{List}}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil})^{\star}\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{array}\right)\ \texttt{True})$$

$$\xrightarrow{no,case}\ (((\lambda u, v.v)^{\star}\ (\lambda w.w))\ \texttt{True})$$

$$\xrightarrow{no,\beta}\quad ((\lambda v.v)^{\star}\ \texttt{True})$$

# Beispiel

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

$$\xrightarrow{no,\beta}\quad ((\lambda y.\left(\begin{array}{l}\texttt{case}_{\texttt{List}}\ y\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{array}\right)\ \texttt{True}))^{\star}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil}))$$

$$\xrightarrow{no,\beta}\quad (\left(\begin{array}{l}\texttt{case}_{\texttt{List}}\ (\texttt{Cons}\ (\lambda w.w)\ \texttt{Nil})^{\star}\ \texttt{of}\ \{\\ \texttt{Nil} \to \texttt{Nil};\\ (\texttt{Cons}\ z\ zs) \to ((\lambda u, v.v)\ z)\}\end{array}\right)\ \texttt{True})$$

$$\xrightarrow{no,case}\ (((\lambda u, v.v)^{\star}\ (\lambda w.w))\ \texttt{True})$$

$$\xrightarrow{no,\beta}\quad ((\lambda v.v)^{\star}\ \texttt{True})$$

$$\xrightarrow{no,\beta}\quad \texttt{True}$$

# Normalordnungsreduktion (2)

**Definition**

Wenn $s$ unmittelbar zu $t$ reduziert, dann ist $R[s] \to R[t]$ für jeden Reduktionskontext $R$ eine Normalordnungsreduktion.

- Notation: $\xrightarrow{no}$, bzw. auch $\xrightarrow{no,\beta}$ und $\xrightarrow{no,case}$.
- $\xrightarrow{no,+}$ transitive Hülle von $\xrightarrow{no}$
- $\xrightarrow{no,*}$ reflexiv-transitive Hülle von $\xrightarrow{no}$

# Normalformen

Ein KFPT-Ausdruck $s$ ist eine

- Normalform (NF = normal form):
  $s$ enthält keine $(\beta)$- oder (case)-Redexe
- Kopfnormalform (HNF = head normal form):
  $s$ ist Konstruktoranwendung oder Abstraktion $\lambda x_1, \ldots x_n.s'$, wobei $s'$ entweder Variable oder $(c\ s_1\ \ldots\ s_{\mathrm{ar}(c)})$ 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 $(c\ s_1\ \ldots\ s_{\mathrm{ar}(c)})$

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

# Beispiel

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

---

$(\texttt{Cons True}\ (\ (\lambda y.y)\ \texttt{Nil}))$ CWHNF

# Normalformen

Warum **schwach?**

# Normalformen

Warum **schwach?**

Schwach: damit es zur Reduktionsstrategie passt:
Strategie ist: Normalordnungsreduktion
D.h. nicht alles in einer WHNF ist ausgewertet.

# Normalformen

Warum **schwach?**

| | |
|---|---|
| Schwach: | damit es zur Reduktionsstrategie passt: |
| | Strategie ist: Normalordnungsreduktion |
| | D.h. nicht alles in einer WHNF ist ausgewertet. |

NF und HNF passen jeweils zu anderen Reduktions-Strategien.

# Terminierung bzw. Konvergenz

**Definition**

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

$$s\Downarrow \iff \exists \text{ WHNF } t : s \xrightarrow{no,*} t$$

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 $\xrightarrow{no,*}$ reduziert werden kann.

# Lambda Kalkül und Typisierung

- Im puren Lambda Kalkül gibt es keine Typisierung.
- Typisierung: wenn man Daten und Funktionen unterscheiden will.
  bzw. wenn man die erlaubten Argumente beschränken will.
- ... und dann geht es weiter ...
- - verschiedene Art von Daten
- - verschiedene Funktionalitäten. ...

# Lambda Kalkül und Typisierung

- Im puren Lambda Kalkül gibt es keine Typisierung.
- Typisierung: wenn man Daten und Funktionen unterscheiden will.
  bzw. wenn man die erlaubten Argumente beschränken will.
- . . . und dann geht es weiter . . .
- - verschiedene Art von Daten
- - verschiedene Funktionalitäten. . . .

# Lambda Kalkül und Typisierung

- Im puren Lambda Kalkül gibt es keine Typisierung.
- Typisierung: wenn man Daten und Funktionen unterscheiden will.
  bzw. wenn man die erlaubten Argumente beschränken will.
- . . . und dann geht es weiter . . .
- - verschiedene Art von Daten
- - verschiedene Funktionalitäten. . . .

# Lambda Kalkül und Typisierung

- Im puren Lambda Kalkül gibt es keine Typisierung.
- Typisierung: wenn man Daten und Funktionen unterscheiden will.
  bzw. wenn man die erlaubten Argumente beschränken will.
- ... und dann geht es weiter ...
- - verschiedene Art von Daten
- - verschiedene Funktionalitäten. ...

# Dynamische Typisierung in KFPT

Normalordnungsreduktion in KFPT stoppt ohne WHNF: Fälle:

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

# Dynamische Typisierung in KFPT

Normalordnungsreduktion in KFPT stoppt ohne WHNF: 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 $T$
- $s = R[\texttt{case}_T\ \lambda x.t\ \texttt{of}\ Alts]$.
- $s = R[(c\ s_1\ \ldots\ s_{\mathrm{ar}(c)})\ t]$

---

# Dynamische Typisierung in KFPT

Normalordnungsreduktion in KFPT stoppt ohne WHNF: 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 $T$
- $s = R[\texttt{case}_T\ \lambda x.t\ \texttt{of}\ Alts]$.
- $s = R[(c\ s_1\ \ldots\ s_{\mathrm{ar}(c)})\ t]$

---

$s$ ist dynamisch ungetypt

$$\iff$$

$$\exists t : s \xrightarrow{no,*} t \wedge t \text{ ist direkt dynamisch ungetypt}$$

# Beispiele

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

# Dynamische Typisierung (2)

**Satz (Progress Lemma)**

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

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

**Fortschrittseigenschaft**

(Progress property):
Wenn $t$ geschlossen, keine WHNF und getypt ist,
dann kann man eine Normalordnungsreduktion auf $t$ durchführen.

Es fehlt noch: Zusammenhang zwischen Typisierung und mehreren Normalordnungs-Reduktionen !?

# Beispiele

Was ist mit $1/0$ in Haskell?

$1/0$ ist in Haskell als externer Aufruf definiert.
Intern würde man es so machen:
case-Ausdruck mit $\Omega$ als Ausgang.
(Wobei $\Omega$ ein Ausdruck ist, der alle Typen hat.)
Dann ist $1/0$ getypt, und Reduktion ergibt Nichtterminierung.

# Darstellung von Ausdrücken als Termgraphen

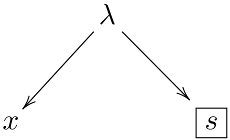
Knoten für je ein syntaktisches Konstrukt des Ausdrucks

- Variablen = ein Blatt

# Darstellung von Ausdrücken als Termgraphen

Knoten für je ein syntaktisches Konstrukt des Ausdrucks

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

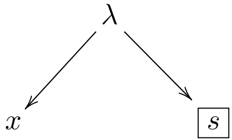



wobei $\boxed{s}$ Baum für $s$

# Darstellung von Ausdrücken als Termgraphen

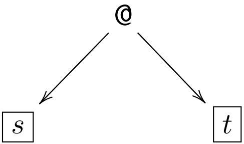
Knoten für je ein syntaktisches Konstrukt des Ausdrucks

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



wobei $\boxed{s}$ Baum für $s$

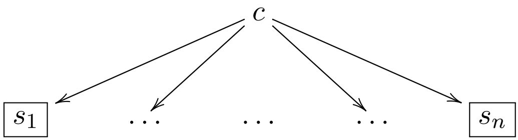
- Applikationen $(s\ t)$



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

# Darstellung von Ausdrücken als Termgraphen (2)

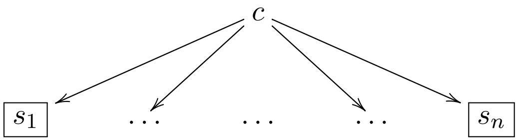
- Konstruktoranwendungen $n$-stellig: $(c\ s_1 \ldots\ s_n)$




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

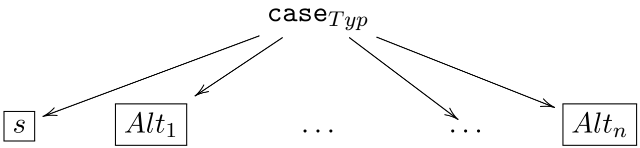
# Darstellung von Ausdrücken als Termgraphen (2)

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




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

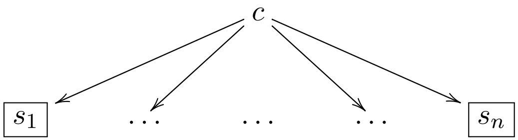
- case-Ausdrücke: $n+1$ Kinder, $\texttt{case}_{Typ}\ s$ `of` $\{Alt_1; \ldots; Alt_n\}$



# Darstellung von Ausdrücken als Termgraphen (2)

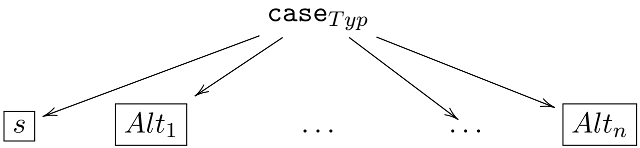


- Konstruktoranwendungen $n$-stellig: $(c\ s_1 \ldots\ s_n)$

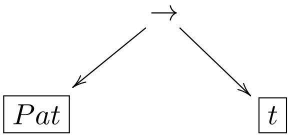



wobei $\boxed{s_i}$ die Bäume für $s_i$

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




- case-Alternative $Pat \to t$

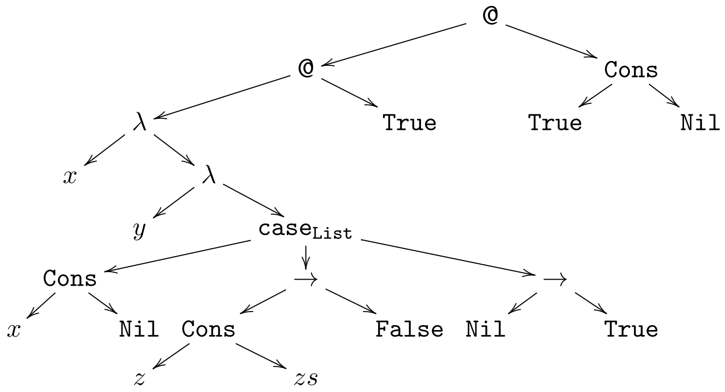


# Beispiel

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

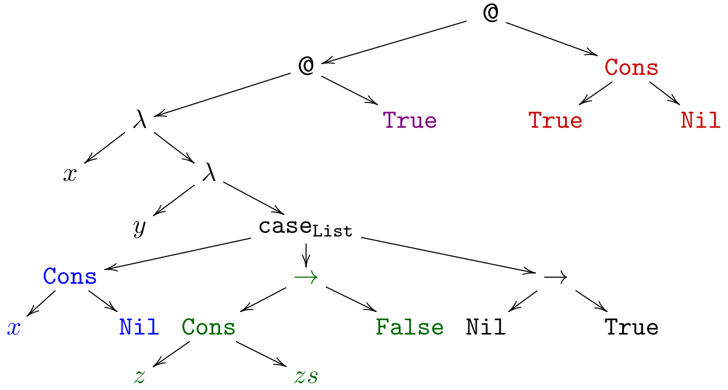
# Beispiel

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



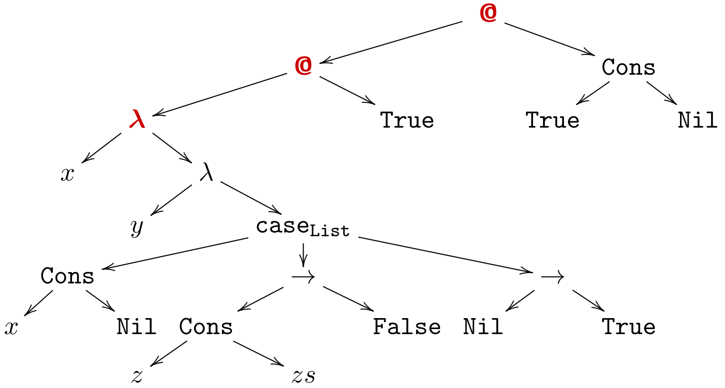
# Beispiel

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



# Beispiel

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




NO-Redex-Suche: immer links, bis Abstraktion oder Konstruktoranwendung

# Normalordnungsreduktion: Eigenschaften

- Die Normalordnungsreduktion ist deterministisch, d.h. für jedes $s$ gibt es höchstens ein $t$ mit $s \xrightarrow{no} t$.
- Eine WHNF ist irreduzibel bezüglich der Normalordnungsreduktion.

# Normalordnungsreduktion: Eigenschaften

- Die Normalordnungsreduktion ist deterministisch, d.h. für jedes $s$ gibt es höchstens ein $t$ mit $s \xrightarrow{no} t$.
- Eine WHNF ist irreduzibel bezüglich der Normalordnungsreduktion.

**Satz (Standardisierung für KFPT)**

*Wenn $s \xrightarrow{*} t$ mit beliebigen $(\beta)$- und $(case)$-Reduktionen (in beliebigem Kontext angewendet), und $t$ ist eine WHNF, dann existiert eine WHNF $t'$, so dass $s \xrightarrow{no,*} t'$ und $t' \xrightarrow{*} t$ (unter $\alpha$-Gleichheit).*

$$\begin{tikzcd}
 & s \arrow[dl, "*"'] \arrow[dr, dashed, "no{,}*"] & \\
t & & t' \arrow[ll, dashed, "*"]
\end{tikzcd}$$

*WHNF* *WHNF*

# KFPT Erweiterung zu KFPTS

KFPTS

# Rekursive Superkombinatoren: KFPTS

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

Annahme: Es gibt eine Menge von Superkombinator**namen** $\mathcal{SK}$.

Beispiele: Superkombinator `length`, `map`, `sum`,

# KFPTS: Syntax

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

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

# KFPTS:Syntax (2)

Zu jedem Superkombinator $SK$
gibt es eine Superkombinatordefinition:

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

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

Beispiel: Superkombinator `length`

$\texttt{length}\ xs = \texttt{case}_{\texttt{List}}\ xs\ \texttt{of}\ \{$
$\quad \texttt{Nil} \rightarrow 0;$
$\quad (\texttt{Cons}\ y\ ys) \rightarrow (1 + \texttt{length}\ ys)\}$

# KFPTS: Syntax (3)

Ein KFPTS-Programm besteht aus:

- einer Menge von Typen und Konstruktoren,
- einer Menge von Superkombinator-Definitionen,
- und aus einem KFPTS-Ausdruck $s$.
  (Diesen könnte man auch als Superkombinator `main` mit Definition $\texttt{main} = s$ definieren.)

Das Programm darf keine freien Variablennamen enthalten: D.h. alle in $s$ verwendeten Superkombinatoren müssen definiert sein.

# KFPTS:Syntax (2) Superkombinatoren

Unterschiede Haskell / KFPTS bei Superkombinatordefinitionen:

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

# KFPTS:Syntax (2) Superkombinatoren

Unterschiede Haskell / KFPTS bei Superkombinatordefinitionen:

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

Ist übersetzbar nach KFPTS.

# Superkombinatoren

Der Begriff Superkombinator im Lambda Kalkül

- geschlossener Ausdruck $s$
- Form $\lambda x_1, \ldots x_n.s'$
- Man kann $s$ durch iterierte Abkürzungen (ohne rekursive Vorkommen) und komplett ohne innere lambda-s darstellen

**Beispiele**

$\lambda x, y.x$

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

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

transformierbar zu: $\lambda x.(x\ ((\lambda z.\lambda y.(y\ z))\ x))$

mittels (interner) Reduktion (rückwärts)

# Superkombinatoren

Der Begriff Superkombinator im Lambda Kalkül

- geschlossener Ausdruck $s$
- Form $\lambda x_1, \ldots x_n.s'$
- Man kann $s$ durch iterierte Abkürzungen
  (ohne rekursive Vorkommen)
  und komplett ohne innere lambda-s darstellen

**Beispiele**

$\lambda x, y.x$

$\lambda x.(x\ ((\lambda y.y)\ x))$ $= \lambda x.(x\ (I\ x))$ mit $I$

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

transformierbar zu: $\lambda x.(x\ ((\lambda z.\lambda y.(y\ z))\ x))$

mittels (interner) Reduktion (rückwärts)

# Superkombinatoren

Der Begriff Superkombinator im Lambda Kalkül

- geschlossener Ausdruck $s$
- Form $\lambda x_1, \ldots x_n.s'$
- Man kann $s$ durch iterierte Abkürzungen
  (ohne rekursive Vorkommen)
  und komplett ohne innere lambda-s darstellen

**Beispiele**

$\lambda x, y.x$

$\lambda x.(x\ ((\lambda y.y)\ x))$ $= \lambda x.(x\ (I\ x))$ mit $I$

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

transformierbar zu: $\lambda x.(x\ ((\lambda z.\lambda y.(y\ z))\ x))$

mittels (interner) Reduktion (rückwärts)

# Superkombinatoren, Einschub eta-Transformation

Die eta ($\eta$)- Transformation im Lambda-Kalkül ist:

$$s \sim_\eta \lambda x.s\ x \quad \text{wenn } x \text{ nicht frei in } s$$

Diese ist in manchen Programmiersprachen korrekt

Vorsicht: Aber i.a. ist diese falsch in Haskell:

- $\perp$ ist keine WHNF, aber $\lambda x.\perp\ x$ ist eine WHNF.
- Fazit: $\eta$ kann die Terminierung von Ausdrücken ändern!!
- $\implies$ wird im Haskell-Compiler nicht verwendet.

# KFPTS: Operationale Semantik

**Reduktionskontexte**:

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

# KFPTS: Operationale Semantik

**Reduktionskontexte**:

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

**Reduktionsregeln** $(\beta)$, (case) und (SK-$\beta$):

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

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

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

# KFPTS: Operationale Semantik

**Reduktionskontexte**:

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

**Reduktionsregeln** ($\beta$), (case) und (SK-$\beta$):

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

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

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

**Normalordnungsreduktion**:

$$\frac{s \to t \qquad \text{mit } (\beta)\text{-, (case)- oder (SK-}\beta)}{R[s] \xrightarrow{no} R[t]}$$

# KFPTS: WHNFs und Dynamische Typisierung

WHNFs

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

Direkt dynamisch ungetypt:

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

# 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-$\beta$)
  - Zu wenig Argumente und kein Kontext außen: WHNF
  - Zu wenig Argumente und im Kontext $(\texttt{case}\ [.]\ \ldots)$: direkt dynamisch ungetypt.

# Beispiel

Die Superkombinatoren $map$ und $not$:

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

# Beispiel

Die Superkombinatoren $map$ und $not$:

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

Beispiel zur Auswertung:

$$
\begin{array}{ll}
 & map\ not\ (\mathtt{Cons}\ \mathtt{True}\ (\mathtt{Cons}\ \mathtt{False}\ \mathtt{Nil})) \\
\xrightarrow{no,\text{SK-}\beta} & \mathtt{case}_{\mathtt{List}}\ (\mathtt{Cons}\ \mathtt{True}\ (\mathtt{Cons}\ \mathtt{False}\ \mathtt{Nil}))\ \mathtt{of}\ \{ \\
 & \qquad \mathtt{Nil} \to \mathtt{Nil}; \\
 & \qquad (\mathtt{Cons}\ y\ ys) \to \mathtt{Cons}\ (not\ y)\ (map\ not\ ys)\} \\
\xrightarrow{no,\text{case}} & \mathtt{Cons}\ (not\ \mathtt{True})\ (map\ not\ (\mathtt{Cons}\ \mathtt{False}\ \mathtt{Nil}))
\end{array}
$$

WHNF erreicht!

Beachte: Im GHCI-Interpreter wird nur aufgrund des Anzeigens in den Argumenten eines Konstruktors weiter ausgewertet

# Erweiterung um seq und strict

- In Haskell gibt es seq:

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

- Operational: Werte erst $a$ aus, dann $b$; und gebe (nur) den Wert von $b$ zurück
- Analog: `strict` (in Haskell infix als `$!` geschrieben)
- `strict f` macht `f` strikt im ersten Argument, d.h. `strict f` wertet erst das Argument aus, dann erfolgt die Definitionseinsetzung.
- `seq` und `strict` sind austauschbar:

```
f $! x    =   seq x (f x)
seq a b   =   (\x -> b) $! a
```

# Erweiterung um seq und strict

- In Haskell gibt es `seq`:

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

- Operational: Werte erst $a$ aus, dann $b$; und gebe (nur) den Wert von $b$ zurück
- Analog: `strict` (in Haskell infix als `$!` geschrieben)
- `strict f` macht `f` strikt im ersten Argument, d.h. `strict f` wertet erst das Argument aus, dann erfolgt die Definitionseinsetzung.
- `seq` und `strict` sind austauschbar:

```
f $! x    =   seq x (f x)
seq a b   =   (\x -> b) $! a
```

# KFPXX+seq Sprachen

Nachweisbar: 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:

$$\texttt{seq}\ v\ t \quad \rightarrow \quad t, \text{ wenn } v \text{ WHNF}$$

erweiterte Reduktionskontexte und die neue Verschieberegel:

$$(\texttt{seq}\ s\ t)^{\star} \quad \rightarrow \quad (\texttt{seq}\ s^{\star}\ t)$$

# Bemerkung: Die (low-level) Sprache KFP

KFP hat keine Typbeschränkungen!

$$\begin{array}{lcl}
\mathbf{Expr} & ::= & V \mid \lambda V.\mathbf{Expr} \mid (\mathbf{Expr}_1\ \mathbf{Expr}_2) \\
 & \mid & (c_i\ \mathbf{Expr}_1\ \ldots\ \mathbf{Expr}_{\mathrm{ar}(c_i)}) \\
 & \mid & (\texttt{case}\ \mathbf{Expr}\ \texttt{of} \\
 & & \quad \{\mathbf{Pat}_1 \to \mathbf{Expr}_1; \ldots; \mathbf{Pat}_n \to \mathbf{Expr}_n; \texttt{lambda} \to \mathbf{Expr}_{n+1}\}) \\
 & & \text{wobei } Pat_1, \ldots, Pat_n \text{ alle Konstruktoren abdecken}
\end{array}$$

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

- Unterschied zu KFPT: Kein getyptes case, lambda-Pattern
- Neue Reduktionsregel case $\lambda x.s$ of $\{\ldots, \texttt{lambda} \to t\} \to t$
- In KFP ist seq kodierbar:

$$\texttt{seq}\ a\ b := \texttt{case}\ a\ \texttt{of}\ \{Pat_1 \to b; \ldots; Pat_n \to b; \texttt{lambda} \to b\}$$

## Typisierung

# Polymorphe Typen

# 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.

Nur Ausdrücke, die einen (polymorphen) Typ besitzen, gehören zu KFPTSP.

# KFPTSP Beispiele

- polymorph: Typen haben Typvariablen
- z.B. `fak` :: `Int` $\rightarrow$ `Int`
- z.B. `map` :: (`a` $\rightarrow$ `b`) $\rightarrow$ (`List a`) $\rightarrow$ (`List b`)
- Haskell: `->` statt $\rightarrow$
- Haskell verwendet `[a]` statt `(List a)`.

# Beispiele

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

# Einige Typregeln

- Für die Anwendung:

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

- Instanziierung

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

- Für case-Ausdrücke:

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

# Beispiel

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

Beide haben Typ: $\texttt{Bool} \to \texttt{Bool} \to \texttt{Bool}$

Mit der Anwendungsregel kann man Typen von Ausdrücken berechnen:

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

# Beispiel

$$\dfrac{\begin{array}{c}\texttt{True} :: \texttt{Bool},\\ \texttt{False} :: \texttt{Bool}\end{array},\quad \dfrac{\dfrac{\dfrac{\texttt{Cons} :: a \to [a] \to [a]}{\texttt{Cons} :: \texttt{Bool} \to [\texttt{Bool}] \to [\texttt{Bool}]},\ \texttt{True} :: \texttt{Bool}}{(\texttt{Cons True}) :: [\texttt{Bool}] \to [\texttt{Bool}]},\quad \dfrac{\texttt{Nil} :: [a]}{\texttt{Nil} :: [\texttt{Bool}]}}{(\texttt{Cons True Nil}) :: [\texttt{Bool}]},\quad \dfrac{\texttt{Nil} :: [a]}{\texttt{Nil} :: [\texttt{Bool}]}}{\texttt{case}_{\texttt{Bool}}\ \texttt{True of}\ \{\texttt{True} \to (\texttt{Cons True Nil}); \texttt{False} \to \texttt{Nil}\} :: [\texttt{Bool}]}$$

# Beispiel

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

# Übersicht

| Kernsprache | Besonderheiten |
|---|---|
| KFP | Erweiterung des call-by-name Lambda-Kalküls um ungetyptes `case` und Datenkonstruktoren, spezielles `case`-Pattern `lambda` ermöglicht Kodierung von `seq`. |
| 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 | KFPTS+seq mit polymorpher Typisierung, sehr geeignete Kernsprache für Haskell |

# Ausblick

- Erörterung der meisten Konstrukte von Haskell
- insbesondere auch: Modulsystem, Typklassen
- Informell: KFPTSP+seq ist die passende Kernsprache
- Genaue Erkärungen und Analysen zu Typisierung und zur Typberechnung kommen noch!
- Exkurs demnächst: Probabilistiche fuktionale Programmierung