

GOETHE
UNIVERSITÄT
FRANKFURT AM MAIN

www.uni-frankfurt.de

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

- 1 Einleitung
- 2 KFPT
- 3 KFPTS
- 4 seq
- 5 Polymorphe Typen

GOETHE
UNIVERSITÄT
FRANKFURT AM MAIN

Einleitung KFPT KFPTS seq Polymorphe Typen

Lambda-Kalkül und Haskell

Lambda-Kalkül und Haskell

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

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

• **Keine echten Daten:**

Zahlen, Boolesche Werte, Listen und komplexe Datenstrukturen fehlen im Lambda-Kalkül.

Ausweg mittels **Church-Kodierung:**

$$\begin{aligned} \text{z.B. } true &= \lambda x, y. x \\ false &= \lambda x, y. y \end{aligned}$$

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

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

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:

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

Aber: Kompliziert und schwer verständlich!

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

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.

Kernsprachen für Haskell

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

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** $\text{ar}(c_i) \in \mathbb{N}_0$
 (ar = „arity“)

Beispiele

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

Haskell-Schreibweise: [] für Nil und : (infix) für Cons
 Beachte $[a_1, a_2, \dots, a_n]$ ist Abkürzung für $a_1 : (a_2 : (\dots : (a_n : [])))$

Syntax von KFPT

$\text{Expr} ::= V$ (Variable)
 $\quad | \lambda V. \text{Expr}$ (Abstraktion)
 $\quad | (\text{Expr}_1 \text{ Expr}_2)$ (Anwendung)
 $\quad | (c_i \text{ Expr}_1 \dots \text{Expr}_{\text{ar}(c_i)})$ (Konstruktoranwendung)
 $\quad | (\text{case}_{\text{Typname}} \text{Expr of } \{ \text{Pat}_1 \rightarrow \text{Expr}_1; \dots; \text{Pat}_n \rightarrow \text{Expr}_n \})$ (case-Ausdruck)
 $\text{Pat}_i ::= (c_i V_1 \dots V_{\text{ar}(c_i)})$ (Pattern für Konstruktor i)
 wobei die Variablen V_i alle verschieden sind.

Nebenbedingungen:

- case mit Typ gekennzeichnet,
- $\text{Pat}_i \rightarrow \text{Expr}_i$ 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):

$\lambda xs. \text{case}_{\text{List}} xs \text{ of } \{ \text{Nil} \rightarrow \perp; (\text{Cons } y \ ys) \rightarrow y \}$

Restliste ohne erstes Element (tail):

$\lambda xs. \text{case}_{\text{List}} xs \text{ of } \{ \text{Nil} \rightarrow \perp; (\text{Cons } y \ ys) \rightarrow ys \}$

\perp repräsentiert Fehler, z.B. Ω

Test, ob Liste leer ist (null):

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

If-Then-Else:

if e then s else t :

$\text{case}_{\text{Bool}} e \text{ of } \{ \text{True} \rightarrow s; \text{False} \rightarrow t \}$

Beispiele (2)

- Paare: Typ Paar mit zweistelligem Konstruktor Paar
Z.B. wird (True, False) durch (Paar True False) dargestellt.
- Projektionen:

$\text{fst} := \lambda x. \text{case}_{\text{Paar}} x \text{ of } \{ (\text{Paar } a \ b) \rightarrow a \}$
 $\text{snd} := \lambda x. \text{case}_{\text{Paar}} x \text{ of } \{ (\text{Paar } a \ b) \rightarrow b \}$

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

Haskell vs. KFPT: case-Ausdrücke (1)

Vergleich mit Haskell's case-Ausdrücken

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

Beispiel:

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

- 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)

- 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 -> []`

Freie und gebundene Variablen in KFPT

Zusätzlich zum Lambda-Kalkül:

In einer case-Alternative

$$(c_i \ x_1 \ \dots \ x_{\text{ar}(c_i)}) \rightarrow s$$

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

Haskell vs. KFPT: case-Ausdrücke (3)

Übersetzung von geschachtelten (Haskell) in einfache Pattern (für KFPT)

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

wird übersetzt in:

```
caseList Nil of { Nil -> Nil;
                 (Cons x z) -> caseList z of { Nil -> ⊥;
                                             (Cons y ys) -> (Cons y Nil)
                                             }
                 }
```

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

Freie Variablen

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

Gebundene Variablen

$$\begin{aligned}
 BV(x) &= \emptyset \\
 BV(\lambda x.s) &= BV(s) \cup \{x\} \\
 BV(s \ t) &= BV(s) \cup BV(t) \\
 BV(c \ s_1 \ \dots \ s_{\text{ar}(c)}) &= BV(s_1) \cup \dots \cup BV(s_{\text{ar}(c)}) \\
 BV(\text{case}_{Typ} \ t \ \text{of} \\
 &\quad \{(c_1 \ x_{1,1} \ \dots \ x_{1,\text{ar}(c_1)}) \rightarrow s_1; \\
 &\quad \dots \\
 &\quad (c_n \ x_{n,1} \ \dots \ x_{n,\text{ar}(c_n)}) \rightarrow s_n \}) \\
 &= BV(t) \cup \left(\bigcup_{i=1}^n (BV(s_i) \cup \{x_{i,1}, \dots, x_{i,\text{ar}(c_i)}\}) \right)
 \end{aligned}$$

Notationen in KFPT

Notationen in KFPT:

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

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

Beispiel

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

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

Alpha-äquivalenter Ausdruck:

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

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

Operationale Semantik von KFPT

Substitution

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

Definition

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

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

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

Wenn $s \rightarrow t$ mit (β) oder (case) dann **reduziert s unmittelbar zu t**

Beispiel

$$\begin{aligned}
 & (\lambda x. \text{case}_{\text{Paar}} x \text{ of } \{(\text{Paar } a \ b) \rightarrow a\}) (\text{Paar True False}) \\
 \xrightarrow{\beta} & \text{case}_{\text{Paar}} (\text{Paar True False}) \text{ of } \{(\text{Paar } a \ b) \rightarrow a\} \\
 \xrightarrow{\text{case}} & \text{True}
 \end{aligned}$$

KFPT-Kontexte

Kontext = Ausdruck mit Loch $[\cdot]$

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

Wenn $C[s] \rightarrow C[t]$ wobei $s \xrightarrow{\beta} t$ oder $s \xrightarrow{\text{case}} t$, 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:

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

Normalordnungsreduktionen finden im Reduktionskontext statt

Redexsuche mit * zum Verschieben

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

Neue Regel:

- $(\text{case}_{\text{Typ}} s \text{ of } \text{Alts})^* \Rightarrow (\text{case}_{\text{Typ}} s^* \text{ of } \text{Alts})$

Beispiel

$$\begin{aligned}
 & (((\lambda x. \lambda y. \left(\begin{array}{l} \text{case}_{\text{List}} y \text{ of } \{ \\ \text{Nil} \rightarrow \text{Nil}; \\ (\text{Cons } z \ zs) \rightarrow (x \ z) \} \end{array} \right) \text{True})) (\lambda u, v. v)) (\text{Cons } (\lambda w. w) \text{ Nil}))^* \\
 \xrightarrow{\text{no}, \beta} & ((\lambda y. \left(\begin{array}{l} \text{case}_{\text{List}} y \text{ of } \{ \\ \text{Nil} \rightarrow \text{Nil}; \\ (\text{Cons } z \ zs) \rightarrow ((\lambda u, v. v) \ z) \} \end{array} \right) \text{True})) (\text{Cons } (\lambda w. w) \text{ Nil}))^* \\
 \xrightarrow{\text{no}, \beta} & \left(\begin{array}{l} \text{case}_{\text{List}} (\text{Cons } (\lambda w. w) \text{ Nil}) \text{ of } \{ \\ \text{Nil} \rightarrow \text{Nil}; \\ (\text{Cons } z \ zs) \rightarrow ((\lambda u, v. v) \ z) \} \end{array} \right) \text{True})^* \\
 \xrightarrow{\text{no}, \text{case}} & (((\lambda u, v. v) (\lambda w. w)) \text{True})^* \\
 \xrightarrow{\text{no}, \beta} & ((\lambda v. v) \text{True})^* \\
 \xrightarrow{\text{no}, \beta} & \text{True}
 \end{aligned}$$

Normalordnungsreduktion (2)

Definition

Wenn s unmittelbar zu t reduziert, dann ist $R[s] \rightarrow R[t]$ 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}

Beispiel

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

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

Normalformen

Ein KFPT-Ausdruck s ist eine

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

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

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 reinen 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.

Definition (Dynamische Typregeln für KFPT)

Ein KFPT-Ausdruck s **direkt dynamisch ungetypt**, falls:

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

s ist **dynamisch ungetypt**

\iff

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

Beispiele

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

(Wobei Ω ein Ausdruck ist, der alle Typen hat.)

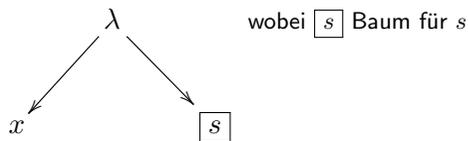
Dann ist 1/0 getypt, und Reduktion ergibt Nichtterminierung.

Darstellung von Ausdrücken als Termgraphen

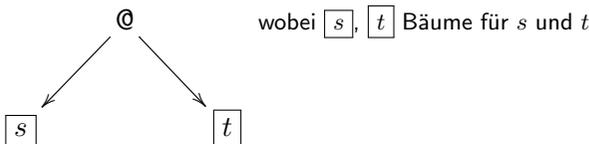
Knoten für je ein syntaktisches Konstrukt des Ausdrucks

- Variablen = ein Blatt

- Abstraktionen $\lambda x.s$



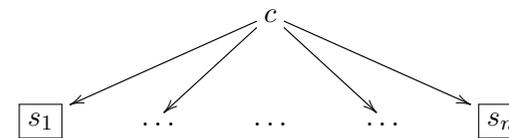
- Applikationen $(s t)$



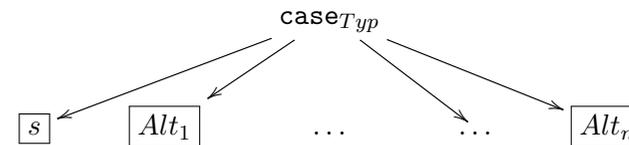
Darstellung von Ausdrücken als Termgraphen (2)

- Konstruktoren n -stellig: $(c s_1 \dots s_n)$

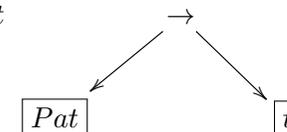
wobei s_i die Bäume für s_i



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



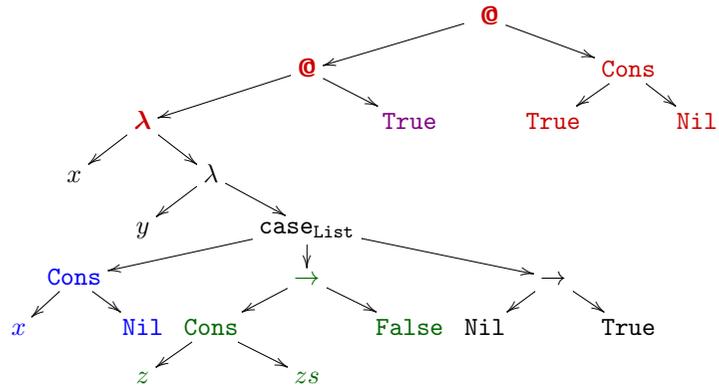
- case-Alternative $Pat \rightarrow t$



Beispiel

```

((
  (
    (
      λx.λy.caseList (Cons x Nil) of {
        (Cons z zs) → False;
        Nil → True
      }
    ) True
  ) (Cons True Nil)
)
    
```



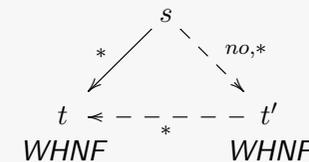
NO-Redex-Suche: immer links, bis Abstraktion oder Konstruktoranwendung

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.

Satz (Standardisierung für KFPT)

Wenn $s \xrightarrow{*} t$ mit beliebigen (β)- 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 α -Gleichheit).



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** SK .

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

KFPTS: Syntax

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

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

KFPTS:Syntax (2)

Zu jedem Superkombinator SK
gibt es eine Superkombinatordefinition:

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

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

Beispiel: Superkombinator length

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

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 `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

Ist übersetzbar nach KFPTS.

Superkombinatoren

Der Begriff **Superkombinator** im Lambda Kalkül

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

Beispiele

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

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

transformierbar zu: $\lambda x. (x ((\lambda z. \lambda y. (y z)) x))$
 mittels (interner) Reduktion (rückwärts)

Superkombinatoren, Einschub eta-Transformation

Die eta (η)- 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: η kann die Terminierung von Ausdrücken ändern!!
- \implies wird im Haskell-Compiler nicht verwendet.

KFPTS: Operationale Semantik

Reduktionskontexte:

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

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

- (β) $(\lambda x. s) t \rightarrow s[t/x]$
- (case) $\text{case}_{Typ} (c s_1 \dots s_{\text{ar}(c)}) \text{ of } \{ \dots; (c x_1 \dots x_{\text{ar}(c)}) \rightarrow t; \dots \}$
 $\rightarrow t[s_1/x_1, \dots, s_{\text{ar}(c)}/x_{\text{ar}(c)}]$
- (SK- β) $(SK s_1 \dots s_n) \rightarrow e[s_1/x_1, \dots, s_n/x_n]$,
 wenn $SK x_1 \dots x_n = e$ die Definition von SK ist

Normalordnungsreduktion:

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

KFPTS: WHNFs und Dynamische Typisierung

WHNFs

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

Direkt dynamisch ungetypt:

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

Markierungsalgorithmus

Markierung funktioniert genauso wie in KFPTS:

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

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.

Erweiterung um seq und strict

- In Haskell gibt es seq:

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

- 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:

$$\begin{aligned} f \ \$! \ x &= \text{seq } x \ (f \ x) \\ \text{seq } a \ b &= (\backslash x \rightarrow b) \ \$! \ a \end{aligned}$$

Beispiel

Die Superkombinatoren `map` und `not`:

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

Beispiel zur Auswertung:

$$\begin{aligned} &\text{map } \text{not} \ (\text{Cons } \text{True} \ (\text{Cons } \text{False} \ \text{Nil})) \\ \xrightarrow{\text{no,SK-}\beta} &\text{case}_{List} \ (\text{Cons } \text{True} \ (\text{Cons } \text{False} \ \text{Nil})) \ \text{of} \ \{ \\ &\quad \text{Nil} \rightarrow \text{Nil}; \\ &\quad (\text{Cons } y \ ys) \rightarrow \text{Cons } (\text{not } y) \ (\text{map } \text{not} \ ys) \} \\ \xrightarrow{\text{no,case}} &\text{Cons } (\text{not } \text{True}) \ (\text{map } \text{not} \ (\text{Cons } \text{False} \ \text{Nil})) \end{aligned}$$

WHNF erreicht!

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

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:

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

erweiterte Reduktionskontexte und die neue Verschieberegeln:

$$(\text{seq } s \ t)^* \rightarrow (\text{seq } s^* \ t)$$

Bemerkung: Die (low-level) Sprache KFP

KFP hat keine Typbeschränkungen!

$\mathbf{Expr} ::= V \mid \lambda V. \mathbf{Expr} \mid (\mathbf{Expr}_1 \mathbf{Expr}_2)$
 $\mid (c_i \mathbf{Expr}_1 \dots \mathbf{Expr}_{ar(c_i)})$
 $\mid (\mathbf{case} \mathbf{Expr} \mathbf{of}$
 $\quad \{\mathbf{Pat}_1 \rightarrow \mathbf{Expr}_1; \dots; \mathbf{Pat}_n \rightarrow \mathbf{Expr}_n; \mathbf{lambda} \rightarrow \mathbf{Expr}_{n+1}\})$
 wobei Pat_1, \dots, Pat_n alle Konstruktoren abdecken

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

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

$\mathbf{seq} a b := \mathbf{case} a \mathbf{of} \{ Pat_1 \rightarrow b; \dots; Pat_n \rightarrow b; \mathbf{lambda} \rightarrow b \}$

Typisierung

Polymorphe Typen

KFPTSP

Mit KFPTSP bezeichnen wir **polymorph getyptes** KFPTS

Definition

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

$\mathbf{T} ::= TV \mid TC \mathbf{T}_1 \dots \mathbf{T}_n \mid \mathbf{T}_1 \rightarrow \mathbf{T}_2$

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. $\mathbf{fak} :: \mathbf{Int} \rightarrow \mathbf{Int}$
- z.B. $\mathbf{map} :: (\mathbf{a} \rightarrow \mathbf{b}) \rightarrow (\mathbf{List} \mathbf{a}) \rightarrow (\mathbf{List} \mathbf{b})$
- Haskell: \rightarrow statt \rightarrow
- Haskell verwendet $[a]$ statt $(\mathbf{List} \mathbf{a})$.

Beispiele

```

True  :: Bool
False :: Bool
not   :: Bool → Bool
map   :: (a → b) → [a] → [b]
(λx.x) :: (a → a)
    
```

Beispiel

```

and := λx,y.caseBool x of {True → y; False → False}
or  := λx,y.caseBool x of {True → True; False → y}
    
```

Beide haben Typ: `Bool → Bool → Bool`

Mit der Anwendungsregel kann man Typen von Ausdrücken berechnen:

$$\frac{\text{and} :: \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}, \text{True} :: \text{Bool}}{(\text{and True}) :: \text{Bool} \rightarrow \text{Bool}}, \text{False} :: \text{Bool}}{(\text{and True False}) :: \text{Bool}}$$

Einige Typregeln

- Für die Anwendung:

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

- Instanziierung

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

- Für case-Ausdrücke:

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

Beispiel

$$\frac{\text{True} :: \text{Bool}, \quad \frac{\text{Cons} :: a \rightarrow [a] \rightarrow [a], \text{True} :: \text{Bool}}{\text{Cons} :: \text{Bool} \rightarrow [\text{Bool}] \rightarrow [\text{Bool}]}, \quad \frac{\text{Nil} :: [a]}{\text{Nil} :: [\text{Bool}]}}{\text{True} :: \text{Bool}, \quad \frac{(\text{Cons True}) :: [\text{Bool}] \rightarrow [\text{Bool}], \quad \text{Nil} :: [\text{Bool}]}{(\text{Cons True Nil}) :: [\text{Bool}]}, \quad \text{Nil} :: [\text{Bool}]}}{\text{False} :: \text{Bool}, \quad \frac{\text{Cons} :: a \rightarrow [a] \rightarrow [a], \text{True} :: \text{Bool}}{\text{Cons} :: \text{Bool} \rightarrow [\text{Bool}] \rightarrow [\text{Bool}]}, \quad \frac{\text{Nil} :: [a]}{\text{Nil} :: [\text{Bool}]}}{\text{case}_{\text{Bool}} \text{True of } \{\text{True} \rightarrow (\text{Cons True Nil}); \text{False} \rightarrow \text{Nil}\} :: [\text{Bool}]}$$

Beispiel

$$\frac{\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]}{\text{map} :: (\text{Bool} \rightarrow \text{Bool}) \rightarrow [\text{Bool}] \rightarrow [\text{Bool}]}, \text{not} :: \text{Bool} \rightarrow \text{Bool}$$

$$(\text{map not}) :: [\text{Bool}] \rightarrow [\text{Bool}]$$

Ausblick

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

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