www.uni-frankfurt.de

JOHANN WOLFGANG GOETHE
UNIVERSITÄT
FRANKFURT AM MAIN

Einführung in die funktionale Programmierung

Prof. Dr. Manfred Schmidt-Schauß

Künstliche Intelligenz und Softwaretechnologie

23. Oktober 2007

Einführung in Haskell: Syntax, Reduktionen, Kernsprachen

JOHANN WOLFGANG GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Haskell, eine nicht-strikte funktionale Programmiersprache mit polymorphem Typsystem.

Ziel dieses Kapitels ist:

- Darstellung der syntaktischen Grundlagen,
- Unterscheidung zwischen Kernbestandteilen der Programmiersprache und zusätzlichen Features,
- Progammierverständnis in Haskell,
- operationale Semantik: Auswertungen

Die Kernsprache KFPT

Die Kernsprache folgt der Lambda-Schreibweise im Lambda-Kalkül, der von Alonzo Church entwickelt wurde.

Der Lambda-Kalkül ist Teil der betrachteten Kernsprache und auch von Haskell.

Einführung der Syntax und Reduktionsregeln

Syntax von KFPT

JOHANN WOLFGANG GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Es gibt *Konstantensymbole*, die jeweils eine feste Stelligkeit haben. Diese nennen wir auch *Konstruktoren*.

Konstruktoren mit Stelligkeit sind anzugeben
(wir erfinden keine KFPT-Syntax dafür)

Es gibt Typen
Jeder Konstruktor gehört zu genau einem Typ.

z.B. seien die Typen `Bool` und `List`; zu `Bool` gehören `True` und `False`, zu `List` gehören `Cons` und `Nil`.
Es gibt pro Typ ein `case`-Konstrukt: $\texttt{case}_{Typname}$.

Syntax von KFPT

Kontextfreie Grammatik für KFPT

Exp ::= V (Variable)
| $(\backslash V$ `->` $Exp)$ wobei V eine Variable ist.
| $(Exp_1\ Exp_2)$
| $(c\ Exp_1\ \ldots\ Exp_n)$ wobei c Konstruktor und $n = ar(c)$
| (case$_{\text{Typ}}$ Exp `of` $\{Pat_1$ `->` $Exp_1; \ldots; Pat_n$ `->` $Exp_n\}$)

Hierbei ist Pat_i Pattern zum Konstruktor i,
Es kommen genau die Konstruktoren zu `Typname` vor
$Pat_i \rightarrow Exp_i$ heißt auch `case`-Alternative.

Pat ::= $(c\ V_1 \ldots V_{ar(c)})$
Die Variablen V_i müssen alle verschieden sein.

Syntax von KFPT

JOHANN WOLFGANG GOETHE UNIVERSITÄT FRANKFURT AM MAIN

$(c\ V_1 \ldots V_{ar(c)})$ sind einfache *Pattern* bzw. *Muster*.

$\lambda x.s$ im Lambda-Ausdrücke entspricht `\x -> s` in KFPT

Zum Beispiel:
$\lambda x.1$ ist die Funktion, die konstant 1 liefert.

Beispiele in KFPT

Die Funktion, die leere Listen erkennt:

```
\xs -> (case_List  xs of  {Nil -> True;  Cons y ys  -> False})
```

In Haskell: kein Typ-Index am `case`,
sondern automatische Typerkennung

Ausdrücke in KFPT (Haskell)

JOHANN WOLFGANG GOETHE UNIVERSITÄT FRANKFURT AM MAIN

- `\x -> s` sind *Abstraktionen* oder Funktionsausdrücke.
- $(s\ t)$ sind *Anwendungen (Applikationen)*.
- $(c\ t_1 \ldots\ t_{ar(c)})$ sind *Konstruktor-Anwendungen*.
- $(\texttt{case}_T\ e\ \texttt{of}\ Alts)$ sind `case`-*Ausdrücke (Fallunterscheidungen)*.

Beispiele und Kodierungen

JOHANN WOLFGANG GOETHE UNIVERSITÄT FRANKFURT AM MAIN

`\x -> x` ist die Identitätsfunktion.

$\texttt{if}\ A\ \texttt{then}\ B\ \texttt{else}\ C$

ist darstellbar durch:

$(\texttt{case}_{\texttt{Bool}}\ A\ \texttt{of}\ \{\texttt{True -> }B; \texttt{False -> }C\})$

Beispiele und Kodierungen

JOHANN WOLFGANG GOETHE
UNIVERSITÄT
FRANKFURT AM MAIN

Sei `Paar` zweistelliger Konstruktor.

`(Paar True 1)` ist ein Objekt zu `Paar`.

Die Selektoren zu `Paar` sind definierbar:

```
\x -> (case_Paar x of  (Paar y1 y2) -> y1)
\x -> (case_Paar x of  (Paar y1 y2) -> y2)
```

Tupel sind vordefiniert in Haskell: Schreibweise ist (x, y, z).

Programm zu Paaren

Hat das Paar p als erstes Element ein `False`?

wird implementiert durch:

```
(case_Paar  p of  {Paar x y  ->
                     (case_Bool x of {True -> False; False -> True}))
```

Listen

Konstruktoren:	`Cons`	mit Stelligkeit 2
	`Nil`	mit Stelligkeit 0

Die Liste `[True, False]` ist darstellbar in KFPT als

```
Cons True (Cons False Nil)
```

Zwei Abkürzungen

JOHANN WOLFGANG GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Id für die identische Abbildung
K für den konstanten Kombinator:

Id	$::=$	`\x -> x`	(Lambda-Notation: $\lambda x.x$)
$K\ x\ y$	$::=$	`\x -> (\y-> x)`	(Lambda-Notation: $\lambda x.(\lambda y.x)$)

Ein nichtterminierender Ausdruck

$$bot := (\lambda x.(x\ x))\ (\lambda y.(y\ y))$$

wir werden später sehen, dass die Auswertung dieses Ausdrucks nicht terminiert.

Freie und gebundene Variablen

JOHANN WOLFGANG GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Die Bindungsregeln in KFPT sind:

- In $\lambda x \ . \ e$ ist x eine gebundene Variable, der Gültigkeitsbereich (Skopus) ist e.

- In der `case`-Alternative $c\ x_1 \ldots x_n$`->` e bindet das Pattern alle Variablen x_i.
  Deren Gültigkeitsbereich (Skopus) ist e.

Vorkommen von Variablen, die in keinem Gültigkeitsbereich einer Bindung stehen, sind *frei*.

Bei Konflikten gilt die Regel: der innerste Bindungsbereich zählt

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

Freie und gebundene Variablen

verschiedene Vorkommen können zu verschiedenen Bindungen gehören:

Beispiel

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

Sichtbar durch Indizes bzw. Umbenennung $\lambda x_1\ .\ (x_1\ (\lambda x_2\ .\ x_2))$

Ausdrücke die bis auf Umbenennung gleich sind,
nennt man auch α-äquivalent
Genauer: durch ein Folge von Umbenennungen
ineinander überführt werden können,

Freie und gebundene Variablen

Ein Ausdruck ohne freie Variablen heißt *geschlossener Ausdruck*

sonst *offener Ausdruck*.

Ein KFPT-*Programm* ist definiert als ein geschlossener KFPT-Ausdruck

Freie und gebundene Variablen. Beispiele


```
\x -> (case_List  x of  {Cons y ys  -> ys; Nil ->  Nil})
```

ist ein geschlossener Ausdruck.

Die Variable x ist frei im Unterausdruck

```
case_List x of {Cons y ys  -> ys; Nil -> Nil}
```

Auswertungsregeln; operationale Semantik

operationale Semantik:

- Transformationssystem auf den Ausdrücken
- deterministisch
- Bei erfolgreichem Terminieren: Der letzte Ausdruck ist der Wert

Operationale Semantik: Wert

Ein *Wert* in KFPT
bzw. eine WHNF (weak head normal form, schwache Kopfnormalform)
ist ein Ausdruck der Form

- $(c\ t_1 \ldots t_n)$, wobei $n = arity(c)$ und c ein Konstruktor ist, oder
- eine Abstraktion: $\lambda x\ .\ e$.

Eine WHNF kann sein:

- *FWHNF*, wenn die WHNF eine Abstraktion ist, oder
- *CWHNF*, wenn die WHNF eine Konstruktoranwendung der Form $(c\ s_1 \ldots s_n)$ ist.

Ersetzungen in Ausdrücken

$$s[t/x]$$

Im Ausdruck s werden die freien Vorkommen der Variablen x durch t ersetzt.

Zu Beachten: kein Einfangen von freien Variablen !

Ausreichende Bedingung: In s gebundene Variablen sind von den freien Variablen in t verschieden
Wenn nicht, dann gebundene Variablen in s umbenennen.

Umbenennung

JOHANN WOLFGANG GOETHE UNIVERSITÄT FRANKFURT AM MAIN

- Abstraktion :
  $\lambda x.e$: $\rightarrow$ $\lambda z.e[z/x]$:
  z sei bisher nicht verwendete Variable.
  In e ersetze alle freien Vorkommen von x durch z.

- `case`-Alternative:
  $c\ x_1 \ldots x_n$`->` e $\rightarrow$ $c\ z_1 \ldots z_n$`->` $e[z_1/x_1, \ldots z_n/x_n]$
  ersetze x_i durch neue Variablen z_i, $i = 1, \ldots, n$
  In e ersetze alle freien Vorkommen von x_i durch z_i:

Beispiel: Einfangen von Variablen

Vorsicht: Auch beim Umbenennen muss man das Einfangen von Variablen vermeiden!

Was ist die richtige Ersetzung in $(\lambda x.\lambda y.(x\ z))[y/z]$?

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

richtig: $(\lambda x.\lambda u.(x\ y))$.
mit Umbenennung:
$(\lambda x.\lambda y.(x\ z))[y/z] \rightarrow (\lambda x.\lambda u.(x\ z))[y/z] \rightarrow (\lambda x.\lambda u.(x\ y))$

Auswertungsregeln

Auswertungsregeln (Reduktionsregeln):

Beta $$\frac{((\lambda x.t)\ s)}{t[s/x]}$$

Case $$\frac{(\texttt{case}_{\mathrm{T}}\ (c\ t_1 \ldots t_n)\ \texttt{of}\ \{\ldots; c\ x_1 \ldots x_n \texttt{->}\ s; \ldots\})}{s[t_1/x_1, \ldots, t_n/x_n]}$$

Zunächst darf man diese überall verwenden:
in allen Programmkontexten

Redex (reducible expression) nennt man
den Unterausdruck s, der unmittelbar reduziert wird.

Beispiele

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

$\longrightarrow$

$(\lambda y.(\lambda z.z)\ (y\ (\lambda z'.z')))$

$(\texttt{case}_{\texttt{List}}\ \texttt{(Cons 1 (Cons 2 Nil))}\ \{\texttt{Cons y ys -> ys; Nil -> Nil}\})$

$\longrightarrow$

(Cons 2 Nil)

Auswertungsregeln

Die Anwendung der Auswertungsregeln ist noch nicht eindeutig:

$(\lambda x\ .x)((\lambda y\ .y)z)$ hat zwei Reduktionsmöglichkeiten.

Ziel: die richtige deterministischen Auswertung

Reduktionskontexte

JOHANN WOLFGANG GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Definition

Reduktionskontexte:

$$R \;::=\; [] \mid (R\ e) \mid (\texttt{case}_T\ R\ \texttt{of}\ \{p_1\texttt{->}\ t_1;\ldots;p_n\texttt{->}\ t_n\})$$

Beobachtung: das Loch eines Reduktionskontextes ist:

nicht in Abstraktionen
nicht in einer Konstruktor-Anwendung
nicht im Argument einer Anwendung.
nicht in case-Alternativen.

Normalordnungs-Reduktion

Normalordnungs-Reduktion (*normal-order-Reduktion*):
Der Ausdruck in einem Reduktionskontext wird reduziert:

$\boxed{R[s] \rightarrow R[t]}$ Ein-Schritt-*Normalordnungsreduktion*

R ein Reduktionskontext,
s keine WHNF
und s reduziert unmittelbar zu t

Der Unterterm s zusammen mit seiner Position
ist der *Normalordnungsredex*

Normalordnungs-Reduktion

Bezeichnungen

$\xrightarrow{n}$	eine Normalordnungsreduktion.
$\xrightarrow{n,+}$ bzw. $\xrightarrow{n,*}$.	die transitive und reflexiv-transitive Hülle von $\xrightarrow{n}$
$\xrightarrow{n,*}$	ist die Normalordnungsrelation oder Auswertung eines Terms.

Terminierung

Sei t geschlossener Term.

Wenn ein t' existiert mit $t \xrightarrow{n,*} t'$ und t' ist WHNF,

dann: t terminiert (auch: konvergiert) Bezeichnung: $t\Downarrow$.

Wenn nicht, dann: t divergiert Bezeichnung: $t\Uparrow$.

Auswertungen

JOHANN WOLFGANG GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Wenn $t\Downarrow$, sagen wir auch: *hat eine WHNF*.

Es gilt: Wenn $t \xrightarrow{n,*} t'$ und t' ist WHNF,
dann ist t' eindeutig

Aber: $t \xrightarrow{*,keine\ n.o.} t''$ und t'' ist WHNF,
kann für viele verschiedene t'' gelten.

Beispiel `(cons` $((\lambda x.x)$ `True)) Nil)` ist WHNF,
aber reduziert zu `(cons True Nil)`
(das ist keine Normalordnung.)

Beispiel zur Normalordnung

$(\lambda x\ .x)((\lambda y\ .y)z)$ hat zwei Reduktionsmöglichkeiten,

aber nur eine Normalordnungsreduktion:

$(\lambda x\ .x)((\lambda y\ .y)z) \xrightarrow{n} ((\lambda y\ .y)z)$.

Der Reduktionskontext R ist: ([] $((\lambda y\ .y)z)$)

Beispiele zur Auswertung

JOHANN WOLFGANG GOETHE UNIVERSITÄT FRANKFURT AM MAIN

$((\lambda x.x)\ \texttt{Nil}) \xrightarrow{n} \texttt{Nil}$

$((\lambda x.\lambda y.x)\ s\ t) \xrightarrow{n} ((\lambda y'.s)\ t) \xrightarrow{n} s.$

Umbenennung $[y'/y]$ nach dem ersten Reduktionsschritt;
y' kommt nicht in s vor

Eigenschaften der Normalordnung

Lemma

- Jede unmittelbare Reduktion in einem Reduktionskontext ist eine Normalordnungsreduktion.
- Der Normalordnungsredex und die Normalordnungsreduktion sind eindeutig.
- Eine WHNF hat keinen Normalordnungsredex und erlaubt keine Normalordnungsreduktion.

Plotkins Sichtweise

Gordon Plotkin:

Eine Programmiersprache hat vier wichtige Komponenten:

- Ausdrücke
- Kontexte
- Werte
- eine Auswertungsrelation auf Ausdrücken.

Das definiert dann u.a. $t\Downarrow$.

Die Theorie der korrekten Transformationen kann man darauf aufbauen:

s, t sind gleich, wenn für alle Kontexte C: $C[s]\Downarrow \Leftrightarrow C[t]\Downarrow$

(siehe zweite Semesterhälfte)

Dynamische Typregeln für KFPT

- Für case:

  ($\texttt{case}_T\ e\ \texttt{of}\ \{alt_1;\ldots;alt_m\}$) ist *direkt dynamisch ungetypt*

  wenn e eine der folgenden Formen hat:

  - $(c\ a_1 \ldots a_{arc})$ und c ist ein Konstruktor, der nicht zum Typ T gehört,
  - $\lambda x\ .\ t$.

- für Konstruktoren:

  $((c\ a_1 \ldots a_{arc})\ e)$ ist *direkt dynamisch ungetypt*.

Dynamische Typregeln für KFPT

JOHANN WOLFGANG GOETHE UNIVERSITÄT FRANKFURT AM MAIN

e ist *dynamisch ungetypt*,
wenn $\exists e' : e \xrightarrow{n,*} e'$ und e' ist direkt dynamisch ungetypt

Typisierung in KFPT

e ist *dynamisch wohlgetypt* wenn e nicht dynamisch ungetypt ist.

Haskells Typisierung ist restriktiver als
“es tritt kein dynamischer Typfehler auf”

Unwind

JOHANN WOLFGANG GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Algorithmus zur Bestimmung des Normalordnungsredex
geschrieben als eine Menge von Regeln,
die ein Label (R) verschieben:

Start mit t^R,

$$\begin{array}{lcl} C[(s\ t)^R] & \rightarrow & C[(s^R\ t)] \\ C[(\texttt{case}\ s\ alts)^R] & \rightarrow & C[(\texttt{case}\ s^R\ alts)] \end{array}$$

Regel-Anwendung solange, bis nicht mehr möglich.

Unwind

JOHANN WOLFGANG GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Danach die Normalordnungs-Reduktion: (falls möglich)

Beta $C[((\lambda x.t)^R\ s)] \quad\to\quad C[t[s/x]]$

Case $C[(\texttt{case}_{\texttt{T}}\ (c\ t_1 \ldots t_n)^R\ \texttt{of}\ \{\ldots;c\ x_1 \ldots x_n\texttt{->}\ s;\ldots\})]$
$\to\quad C[s[t_1/x_1,\ldots,t_n/x_n]]$

wenn keine möglich ist, dann gibt es zwei Möglichkeiten:
WHNF oder dynamischer Typfehler.

WHNF: genau dann, wenn R-Label des Top-Terms sich durch die Regeln nicht verschieben lässt.

Dynamisches Verhalten der Normalordnung

Normalordnung reduziert einen R-markierten Unterterm,

bis dieser in WHNF ist.

Dies ergibt einen rekursiven Auswertungsalgorithmus

(leider einen ineffizienten, den man so nicht implementiert)

Beispiel: Auswertung

bot := $(\lambda x.(x\ x))\ (\lambda y.(y\ y))$ terminiert nicht:

Die (beta)-Reduktion ergibt:

$$(\lambda x.(x\ x))^R\ (\lambda y.(y\ y)) \xrightarrow{n} ((\lambda y.(y\ y))\ (\lambda y.(y\ y))) \xrightarrow{n} \ldots$$

Beispiel: Boolesche Funktionen

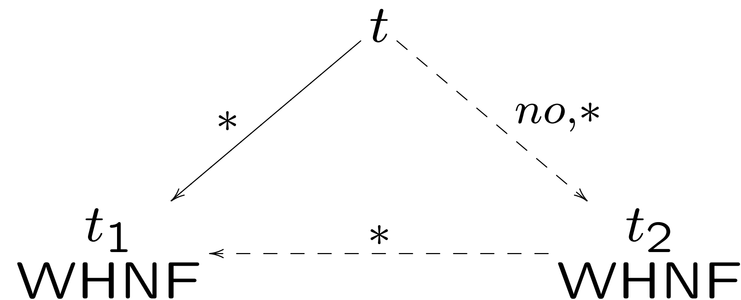
aussagenlogische Verknüpfungen `und`, `oder`, `nicht`:

$$
\begin{array}{lcl}
\texttt{und} & = & \lambda x\ .\ \lambda y\ .\ \texttt{case}_{\texttt{Bool}}\ x\ \texttt{of}\ \{\texttt{True->}\ y;\texttt{False->}\ \texttt{False}\} \\
\texttt{oder} & = & \lambda x\ .\ \lambda y\ .\ \texttt{case}_{\texttt{Bool}}\ x\ \texttt{of}\ \{\texttt{True->}\ \texttt{True};\texttt{False->}\ y\} \\
\texttt{nicht} & = & \lambda x\ .\ \texttt{case}_{\texttt{Bool}}\ x\ \texttt{of}\ \{\texttt{True->}\ \texttt{False};\texttt{False->}\ \texttt{True}\}
\end{array}
$$

In Haskell sind die logischen Verknüpfungen bereits im Prelude definiert als `&&`, `||`, `not`.

Standardisierungs-Satz

Satz (Standardisierung). Sei t ein (evtl. offener) Ausdruck und $t \xrightarrow{*} t_1$ mit (Beta) und (case)-Reduktionen, wobei t_1 eine WHNF ist.
Dann existiert eine WHNF t_2, so dass $t \xrightarrow{no,*} t_2$, und $t_2 \xrightarrow{*} t_1$.



Folgerung: es ist korrekt, an beliebigen Positionen zu reduzieren, bis eine WHNF erreicht ist.

Wenn die erreichte WHNF eine Konstruktorkonstante ist, dann ist sie sogar eindeutig.

Syntax der funktionalen Kernsprache KFP

Wir geben die Syntax von KFP hier zur Information schon mal an.
Sie wird in der zweiten Semesterhälfte genauer besprochen

KFP ist analog zu Kernsprachen von Compilern funktionaler Programmiersprachen.

Einfache Syntax, möglichst wenig vordefiniert

Syntax der funktionalen Kernsprache KFP

Syntax: von KFP:
Es gibt N *Konstruktoren* $c_i, i = 1, \ldots, N$
mit jeweils Stelligkeit $ar(c_i)$

CFG für KFP-Ausdrücke:

$$
\begin{array}{lll}
\text{EXP} & ::= & V \qquad V \text{ sind Variablen} \\
& & \mid \lambda V \,.\, EXP \\
& & \mid (EXP\ EXP) \\
& & \mid (c\ EXP_1\ \ldots\ EXP_n) \qquad \text{wobei } c \text{ Konstruktor und } n = ar(c) \\
& & \mid (\texttt{case}\ EXP\ \{Pat_1 \rightarrow Exp_1; \ldots; Pat_{N+1} \rightarrow Exp_{N+1}\}) \\
& & \quad Pat_i \text{ ist Pattern zum Konstruktor } i \text{, und} \\
& & \quad Pat_{N+1} \text{ das Pattern } \texttt{lambda}. \\
\text{Pat} & ::= & (c\ V_1 \ldots V_{ar(c)}) \mid \texttt{lambda}
\end{array}
$$

KFP und KFPT

KFP: Unterschied zu KFPT:

- KFP hat ein ungetyptes `case`-Konstrukt
- zu jedem Konstruktoren im `case` ex. eine Alternative
- Abstraktionen haben Pattern: `lambda`

⇒: Dadurch kann man `seq` in KFP definieren

Rekursive Superkombinatoren: KFPTS

Wir erweitern KFPT um Haskells rekursive Superkombinatoren.

Die Erweiterung nennen wir KFPTS

$\Rightarrow$: Programmierung in KFPTS wie in einem vereinfachten Haskell

KFPTS Syntax

JOHANN WOLFGANG GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Ausdrücke wie in KFPT, nur um Superkombinatornamen erweitert.

Es gibt Funktionsnamen (Superkombinatornamen) als neue Konstanten und eine Menge von Definitionen der Form

Superkombinatorname $V_1 \ldots V_n = Exp$

Bedingungen:

- $V_1 \ldots V_n$ sind verschieden;
  In Exp kommen nur diese Variablen frei vor.
- Die Namensräume für Variablen, Konstruktoren, Kombinatornamen sind disjunkt.
- Jeder Superkombinator wird höchstens einmal definiert

Das eröffnet die Möglichkeit, rekursive Funktionen zu definieren da Namen von Funktionen beliebig verwendet werden können.

KFPTS-Programme

Ein KFPTS-*Programm* besteht aus:

1. Einer Menge von Typen und Konstruktoren wie in KFPT,
2. einer Menge von Kombinator-Definitionen
3. und aus einem Ausdruck.
   definiert als $\boxed{\texttt{main} = \; Exp}$ (`main`-Konvention)

KFPTS-Auswertung

Reduktionsregeln:

Beta $\dfrac{((\lambda x.t)\ s)}{t[s/x]}$

SK-Beta $\dfrac{(\mathtt{f}\ a_1 \ldots\ a_n)}{r[a_1/x_1, \ldots, a_n/x_n]}$ wenn f definiert ist als: $\mathtt{f}\ x_1\ \ldots\ x_n = r$
d.h. wenn $ar(\mathtt{f}) = n$

Case $\dfrac{(\mathtt{case_T}\ (c\ t_1 \ldots t_n)\ \mathtt{of}\ \{\ldots; c\ x_1 \ldots x_n \texttt{->}\ s; \ldots\})}{s[t_1/x_1, \ldots, t_n/x_n]}$
wobei $n = ar(c)$ und c ist T-Konstruktor

Unterschied zu Beta in KFPT:

SK-Beta reduziert nur bei ausreichend vielen Argumenten.

Beispiel

$$
\begin{array}{lcl}
\texttt{map}\ f\ xs & = & \texttt{case}_{\texttt{List}}\ xs\ \texttt{of}\ \{(\texttt{Cons}\ y\ ys) \rightarrow (\texttt{Cons}\ (f\ y)\ (\texttt{map}\ f\ ys)); \\
 & & \qquad\qquad\qquad\qquad \texttt{Nil} \rightarrow \texttt{Nil}\} \\
\texttt{main} & = & \texttt{map not (Cons True Nil)}
\end{array}
$$

KFPTS-Normal-Ordnung

Reduktionskontexte in KFPTS wie in KFPT

Der Wert des Programms
ist der Wert des `main`-Ausdrucks nach Auswertung.

KFPTS-Normalordnungsreduktion:
innerhalb eines KFPTS-Reduktionskontexts wird
unmittelbar mit Case, Beta oder SK-Beta reduziert.

KFPTS-Normal-Ordnung

JOHANN WOLFGANG GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Normal-Ordnungs Reduktionen mit R-Label:

Beta $C[((\lambda x.t)^R\ s)] \quad \rightarrow \quad C[t[s/x]]$

Case $C[(\texttt{case}_{\mathrm{T}}\ (c\ t_1 \ldots t_n)^R\ \texttt{of}\ \{\ldots ; c\ x_1 \ldots x_n \texttt{->}\ s; \ldots\})]$
$\rightarrow \quad C[s[t_1/x_1, \ldots, t_n/x_n]]$

Neu :

SK-Beta $C[(SK^R\ s_1 \ldots s_{ar(SK)})] \rightarrow C[r_{SK}[s_1/V_1, \ldots, s_n/V_n]]$

KFPTS-WHNF

Eine KFPTS-WHNF ist entweder

- ein Ausdruck $(c\ t_1 \ldots t_n)$; oder
- ein Ausdruck $(f\ t_1 \ldots t_n)$, wobei f Kombinator und $n < ar(f)$; oder
- ein Ausdruck $\lambda x.s$

Beispiel: `map` und `(map not)` sind WHNFs

strict und seq

(strict f t) macht f strikt im ersten Argument:

$$\frac{\text{Wert}(f\ t) = a;\ t\Downarrow}{\text{Wert}(\texttt{strict}\ f\ t) = a} \quad \text{und} \quad \frac{t\Uparrow}{(\texttt{strict}\ f\ t)\Uparrow}$$

(seq s t) wertet s und t aus und gibt den Wert von t zurück:

$$\frac{\text{Wert}(t) = a;\ s\Downarrow}{\text{Wert}(\texttt{seq}\ s\ t) = a} \quad \text{und} \quad \frac{s\Uparrow}{(\texttt{seq}\ s\ t)\Uparrow}$$

strict und seq

Beide sind gleichwertig:

strict f t = seq t (f t)   seq s t = strict (\\x->t) s

In KFP sind strict und seq definierbar:

```
strict f  = \x . case x of {p_1 -> f x; ... ;p_N -> f x;p_{N+1} -> f x}
```

Beachte: In KFPT, KFPTS sind strict und seq nicht definierbar.

Betrachtete Kernsprachen

JOHANN WOLFGANG GOETHE UNIVERSITÄT FRANKFURT AM MAIN

KFP	Abstraktionen, Anwendungen,Konstruktoranwendungen jedes `case` hat alle Pattern und ein extra Pattern `lambda`
KFPT	Abstraktionen, Anwendungen,Konstruktoranwendungen jedes `case` hat nur Pattern eines Typs
KFPTS	Wir KFPT; Superkombinatoren sind erlaubt.
KFPT+ `seq`	KFPT und `seq` wird als definiert angenommen
KFPTS+ `seq`	KFPTS und `seq` wird als definiert angenommen
KFPTSP	KFPTS + `seq` und polymorphe Typisierung