

Semantik funktionaler Programmiersprachen: Funktionale Kernsprachen

Prof. Dr. David Sabel

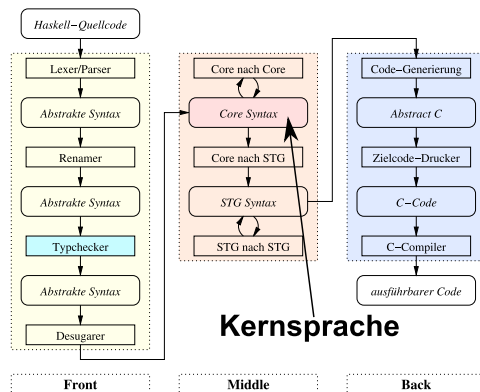
LFE Theoretische Informatik



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

Einleitung

Compilerphasen des GHC (schematisch)



Überblick

- Als Ziel: Definition einer Kernsprache, die zu Haskell passt
- Vorgehen:
 - Wir betrachten zunächst den **Lambda-Kalkül**
 - Er ist „Kernsprache“ fast aller (funktionalen) Programmiersprachen
 - Allerdings oft zu minimalistisch
 - Betrachte **Erweiterte Lambda-Kalküle**
- Kalkül: **Syntax** und **Semantik**
- Sprechweise: **Der** Kalkül (da mathematisch)

Syntax

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

Semantik

- Legt die **Bedeutung** der Programme fest
- Gebiet der **formalen Semantik** kennt verschiedene Ansätze
→ kurzer Überblick auf den nächsten Folien

Axiomatische Semantik

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

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

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

Ansätze zur Semantik (2)

Denotationale Semantik

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

$$\llbracket \text{if } a \text{ then } b \text{ else } c \rrbracket = \begin{cases} \llbracket b \rrbracket, & \text{falls } \llbracket a \rrbracket = \text{True} \\ \llbracket c \rrbracket, & \text{falls } \llbracket a \rrbracket = \text{False} \\ \perp, & \text{sonst} \end{cases}$$

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

Ansätze zur Semantik (3)

Operationale Semantik

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

Der Lambda-Kalkül

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

Der Lambda-Kalkül

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

Syntax des Lambda-Kalküls

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

- Abstraktionen sind **anonyme Funktionen**
Z. B. $id(x) = x$ in Lambda-Notation $\lambda x.x$
- Haskell: $\backslash x \rightarrow s$ statt $\lambda x.s$
- $(s t)$ erlaubt die Anwendung von Funktionen auf Argumente
 s, t beliebige Ausdrücke \implies Lambda Kalkül ist **higher-order**
Z. B. $(\lambda x.x) (\lambda x.x)$ (entspricht gerade $id(id)$)

Syntax des Lambda-Kalküls (2)

Assoziativitäten, Prioritäten und Abkürzungen

- Klammerregeln: $s r t$ entspricht $(s r) t$
- Priorität: Rumpf einer Abstraktion so groß wie möglich:
 $\lambda x.x y$ ist $\lambda x.(x y)$ und **nicht** $((\lambda x.x) y)$
- $\lambda x, y, z.s$ entspricht $\lambda x.\lambda y.\lambda z.s$

Syntax des Lambda-Kalküls (3)

Prominente Ausdrücke

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

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

Freie und gebundene Vorkommen von Variablen

Durch λx ist x im Rumpf s von $\lambda x.s$ **gebunden**.

Kommt x in t vor, so

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

Beispiel:

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

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

Freie und gebundene Variablen

Menge der freien und gebundenen Variablen

$FV(t)$: Freie Variablen von t

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

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

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

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

t ist **geschlossen** bzw. t ist ein **Programm**

Anderenfalls: t ist ein **offener** Ausdruck

$$\begin{aligned} \text{Z.B. } BV(\lambda x.(x (\lambda z.(y z)))) &= \{x, z\} \\ FV(\lambda x.(x (\lambda z.(y z)))) &= \{y\} \end{aligned}$$

Substitution

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

Formale Definition

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

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

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

Kontexte

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

Als Grammatik

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

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

Beispiel zur DVC und α -Umbenennung

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

\implies erfüllt die DVC nicht.

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

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

Alpha-Äquivalenz

Alpha-Umbenennungsschritt

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

Alpha-Äquivalenz

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

- Wir betrachten α -äquivalente Ausdrücke als **gleich**.
- z.B. $\lambda x. x =_{\alpha} \lambda y. y$
- **Distinct Variable Convention**: Alle gebundenen Variablen sind verschieden und gebundene Variablen sind verschieden von freien.
- α -Umbenennungen ermöglichen, dass die DVC stets erfüllt werden kann.

Operationale Semantik - Beta-Reduktion

Beta-Reduktion

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

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

Beispiele:

$$\begin{aligned} & (\lambda x. \underbrace{x}_s) (\underbrace{\lambda y. y}_t) \xrightarrow{\beta} x[(\lambda y. y)/x] = \lambda y. y \\ & (\lambda y. \underbrace{y y}_s) (\underbrace{x z}_t) \xrightarrow{\beta} (y y y)[(x z)/y] = (x z) (x z) (x z) \end{aligned}$$

Beta-Reduktion: Umbenennungen

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

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

Call-by-Name-Reduktion

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

Definition

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

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

ein **Call-by-Name-Reduktionsschritt**

$$\begin{aligned} \text{Beispiel: } & \xrightarrow{\beta} ((\lambda x.(x x)) (\lambda y.y)) ((\lambda w.w) (\lambda z.(z z))) \\ &= (x x)[(\lambda y.y)/x] ((\lambda w.w) (\lambda z.(z z))) \\ &= ((\lambda y.y) (\lambda y.y)) ((\lambda w.w) (\lambda z.(z z))) \\ &R = ([\cdot] ((\lambda w.w) (\lambda z.(z z)))) \end{aligned}$$

Operationale Semantik

- Für die Festlegung der operationalen Semantik muss man noch definieren, **wo** in einem Ausdruck die β -Reduktion angewendet wird

- Betrachte $((\lambda x.x x)((\lambda y.y)(\lambda z.z)))$.

Zwei Möglichkeiten:

- $((\lambda x.x x)((\lambda y.y)(\lambda z.z))) \rightarrow ((\lambda y.y)(\lambda z.z)) ((\lambda y.y)(\lambda z.z))$ oder
- $((\lambda x.x x)((\lambda y.y)(\lambda z.z))) \rightarrow ((\lambda x.x x)(\lambda z.z))$.

Reduktionskontexte: Beispiele

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

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

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

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

- $R = ([\cdot] ((\lambda z.(\lambda x.x) z) u))$, Term t ist $((\lambda w.w) (\lambda y.y))$

Reduktion möglich: $((\lambda w.w) (\lambda y.y)) \xrightarrow{\beta} (\lambda y.y)$.

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

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

Redexsuche: Markierungsalgorithmus

- s ein Ausdruck.
- Start: s^*
- Verschiebe-Regel

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

so oft anwenden wie möglich.

- Beispiel 1: $(((\lambda x.x) (\lambda y.(y y))) (\lambda z.z))^*$
- Beispiel 2: $((y z) ((\lambda w.w)(\lambda x.x))(\lambda u.u))^*$

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

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

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

Beispiel

$$\begin{aligned} & (((\lambda x.\lambda y.x)((\lambda w.w)(\lambda z.z)))(\lambda u.u))^* \\ \Rightarrow & (((\lambda x.\lambda y.x)((\lambda w.w)(\lambda z.z)))^*(\lambda u.u)) \\ \Rightarrow & (((\lambda x.\lambda y.x)^*((\lambda w.w)(\lambda z.z)))(\lambda u.u)) \end{aligned}$$

$$\begin{aligned} \xrightarrow{\text{name}} & ((\lambda y.((\lambda w.w)(\lambda z.z)))(\lambda u.u))^* \\ \Rightarrow & ((\lambda y.((\lambda w.w)(\lambda z.z)))^*(\lambda u.u)) \end{aligned}$$

$$\begin{aligned} \xrightarrow{\text{name}} & ((\lambda w.w)(\lambda z.z))^* \\ \Rightarrow & ((\lambda w.w)^*(\lambda z.z)) \end{aligned}$$

$$\xrightarrow{\text{name}} (\lambda z.z)$$

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

Die Call-by-Name-Reduktion ist deterministisch:

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

Ausdrücke, für die keine Reduktion möglich ist:

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

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

Weitere Notationen:

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

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

Definition

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

Anmerkungen

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

Aussage (Standardisierung)

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

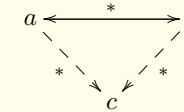
Church-Rosser Theorem

Für den Lambda-Kalkül gilt (unter der Gleichheit $=_a$)

Satz (Konfluenz)

Church-Rosser Eigenschaft:

Wenn $a \leftrightarrow^* b$, dann existiert c , so dass $a \xrightarrow{*} c$ und $b \xrightarrow{*} c$



Hierbei bedeutet:

$\xrightarrow{*}$ beliebige Folge von β -Reduktionen (in beliebigem Kontext), und

\leftrightarrow^* beliebige Folge von β -Reduktionen (vorwärts und rückwärts) (in beliebigem Kontext)

Implementierung eines Interpreters für den Lambda-Kalkül (1)

Datentyp für Ausdrücke:

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

Polymorph über den Namen der Variablen, wir werden im folgenden Strings hierfür verwenden.

Beispiel $s = (\lambda x.x) (\lambda y.y)$:

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

Implementierung eines Interpreters für den Lambda-Kalkül (2)

Funktion `rename`: Umbenennung eines Ausdrucks mit frischen Variablennamen

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

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

Implementierung eines Interpreters für den Lambda-Kalkül (3)

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

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

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

Implementierung eines Interpreters für den Lambda-Kalkül (4)

Ein Call-by-Name-Reduktionsschritt, falls möglich

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

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

-- Andere Faelle: Keine Reduktion möglich:
tryNameBeta _ vars = Nothing
```

Implementierung eines Interpreters für den Lambda-Kalkül (5)

Auswerten, solange wie möglich:

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

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

Implementierung eines Interpreters für den Lambda-Kalkül (6)

Beispielausdrücke:

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

Beispielaufufe:

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

Call-by-Value-Reduktion

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

Call-by-value Beta-Reduktion

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

Definition

CBV-Reduktionskontexte E werden erzeugt durch **ECtxt**:

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

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

Beispiel

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

CBV-Redexsuche mit Markierungsalgorithmus

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

- Beispiel: $((((\lambda x.x) ((\lambda y.y) v) (\lambda z.z))) u) (\lambda w.w))^*$

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

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

Eigenschaften

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

Definition

Ein Ausdruck s **call-by-value konvergiert** ($s \downarrow_{value}$), gdw.

\exists FWHNF $v : s \xrightarrow{value,*} v$.

Ansonsten (call-by-value) **divergiert** s , Notation: $s \uparrow_{value}$.

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

Eigenschaften

Vorteile der Call-by-Value-Reihenfolge:

- Tlw. besseres Platzverhalten
- Auswertungsreihenfolge liegt fest (im Syntaxbaum von f); bzw. ist vorhersagbar.
Bei CBV: $f s_1 s_2 s_3$: immer zuerst s_1 , dann s_2 , dann s_3 , dann $(f s_1 s_2 s_3)$
Bei CBN: zum Beispiel: zuerst s_1 , dann evtl. abhängig vom Wert von s_1 :
zuerst s_2 , dann s_3 , oder andersrum,
oder evtl. weder s_2 noch s_3 .
Dazwischen auch Auswertung von Anteilen von $(f s_1 s_2 s_3)$.
- Wegen der vorhersagbaren Auswertungsreihenfolge unter CBV:
Seiteneffekte können unter CBV direkt eingebaut werden

Beispiele

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

In Haskell: seq

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

$$\begin{aligned} \text{seq } a \ b &= b \quad \text{falls } a \downarrow \\ (\text{seq } a \ b) \uparrow &\quad \text{falls } a \uparrow \end{aligned}$$

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

Verzögerte Auswertung

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

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

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

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

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

Reduktionskontexte \mathbf{R}_{need} :

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

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

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

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

$$\begin{aligned}(\text{lbeta}) \quad &R_{need}[(\lambda x.s) t] \rightarrow R_{need}[\text{let } x = t \text{ in } s] \\ (\text{cp}) \quad &LR[\text{let } x = \lambda y.s \text{ in } R_{need}[x]] \rightarrow LR[\text{let } x = \lambda y.s \text{ in } R_{need}[\lambda y.s]] \\ (\text{llet}) \quad &LR[\text{let } x = (\text{let } y = s \text{ in } t) \text{ in } R_{need}[x]] \\ &\rightarrow LR[\text{let } y = s \text{ in } (\text{let } x = t \text{ in } R_{need}[x])] \\ (\text{lapp}) \quad &R_{need}[(\text{let } x = s \text{ in } t) r] \rightarrow R_{need}[\text{let } x = s \text{ in } (t r)]\end{aligned}$$

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

Markierungsalgorithmus zur Redexsuche (1)

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

Verschiebe-Regeln:

$$\begin{aligned}(1) \quad &(\text{let } x = s \text{ in } t)^\star \quad \Rightarrow \quad (\text{let } x = s \text{ in } t^\star) \\ (2) \quad &(\text{let } x = C_1[y^\diamond] \text{ in } C_2[x^\odot]) \quad \Rightarrow \quad (\text{let } x = C_1[y^\diamond] \text{ in } C_2[x]) \\ (3) \quad &(\text{let } x = s \text{ in } C[x^{\star \vee \diamond}]) \quad \Rightarrow \quad (\text{let } x = s^\diamond \text{ in } C[x^\odot]) \\ (4) \quad &(s t)^{\star \vee \diamond} \quad \Rightarrow \quad (s^\diamond t)\end{aligned}$$

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

Markierungsalgorithmus zur Redexsuche (2)

Reduktion nach Markierung:

$$\begin{aligned}(\text{lbeta}) \quad &((\lambda x.s)^\diamond t) \rightarrow \text{let } x = t \text{ in } s \\ (\text{cp}) \quad &\text{let } x = (\lambda y.s)^\diamond \text{ in } C[x^\odot] \rightarrow \text{let } x = \lambda y.s \text{ in } C[\lambda y.s] \\ (\text{llet}) \quad &\text{let } x = (\text{let } y = s \text{ in } t)^\diamond \text{ in } C[x^\odot] \\ &\rightarrow \text{let } y = s \text{ in } (\text{let } x = t \text{ in } C[x]) \\ (\text{lapp}) \quad &((\text{let } x = s \text{ in } t)^\diamond r) \rightarrow \text{let } x = s \text{ in } (t r)\end{aligned}$$

Beispiel: Call-by-Need-Reduktion

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

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

$$\begin{aligned} & \text{let } x_1 = s_1 \text{ in} \\ & (\text{let } x_2 = s_2 \text{ in} \\ & (\dots \\ & (\text{let } x_n = s_n \text{ in } \lambda x.s))) \end{aligned}$$

Konvergenz und Eigenschaften

Definition

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

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

Satz

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

Gleichheit

Kalküle bisher:

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

D.h. Syntax + Operationale Semantik.

Es fehlt:

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

Gleichheit (2)

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

$C[s]$ und $C[t]$ verhalten sich gleich.

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

Gleichheit (3)

Kontextuelle Approximation und Gleichheit

Call-by-Name Lambda-Kalkül:

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

Call-by-Value Lambda-Kalkül:

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

Gleichheit (4)

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

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

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

Gleichheit (5)

Beispiele für Gleichheiten:

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

Lambda-Kalkül und Haskell

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

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

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

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

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

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

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

Datentypen

Annahmen:

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

Beispiele

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

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

Beispiele (1)

Erstes Element einer Liste (head):

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

Restliste ohne erstes Element (tail):

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

\perp repräsentiert Fehler, z.B. Ω

Test, ob Liste leer ist (null):

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

If-Then-Else:

if e then s else t :

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

Syntax von KFPT

$$\begin{aligned} \text{Expr} ::= & V && \text{(Variable)} \\ & | \lambda V. \text{Expr} && \text{(Abstraktion)} \\ & | (\text{Expr}_1 \text{ Expr}_2) && \text{(Anwendung)} \\ & | (c_i \text{ Expr}_1 \dots \text{Expr}_{ar(c_i)}) && \text{(Konstruktoranwendung)} \\ & | (\text{case}_{\text{Typname}} \text{Expr of} \\ & \quad \{\text{Pat}_1 \rightarrow \text{Expr}_1; \dots; \text{Pat}_n \rightarrow \text{Expr}_n\}) && \text{(case-Ausdruck)} \end{aligned}$$
$$\text{Pat}_i ::= (c_i V_1 \dots V_{ar(c_i)}) \quad \text{(Pattern für Konstruktor } i)$$

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 (2)

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

- Projektionen:

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

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

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

- Haskell erlaubt **überlappende Pattern** und **geschachtelte Pattern**. Z.B. ist $\text{case } [] \text{ of } \{\ [] \rightarrow []; (x:(y:ys)) \rightarrow [y]; x \rightarrow []\}$ ein gültiger Haskell-Ausdruck
- Semikolon und Klammern kann man bei Einrückung weglassen:
 $\text{case } [] \text{ of}$
 - $\ [] \rightarrow []$
 - $\ (x:(y:ys)) \rightarrow [y]$
 - $\ x \rightarrow []$

Haskell vs. KFPT: case-Ausdrücke (3)

Übersetzung von geschachtelten in einfache Pattern (für KFPT)

$\text{case } [] \text{ of } \{\ [] \rightarrow []; (x:(y:ys)) \rightarrow [y]\}$

wird übersetzt in:

$$\text{case}_{List} Nil \text{ of } \{\text{Nil} \rightarrow \text{Nil};$$
$$\quad (\text{Cons } x \text{ } z) \rightarrow \text{case}_{List} z \text{ of } \{\text{Nil} \rightarrow \perp;$$
$$\quad \quad (\text{Cons } y \text{ } ys) \rightarrow (\text{Cons } y \text{ } Nil)$$
$$\quad \quad \quad \}$$
$$\quad \quad \quad \}$$

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

Freie und gebundene Variablen in KFPT

Zusätzlich zum Lambda-Kalkül: In case-Alternative

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

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

Freie Variablen

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

Gebundene Variablen

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

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:

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

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

Notationen

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

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

Operationale Semantik

Substitution

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

$$\begin{aligned}
 (\beta) \quad & (\lambda x. s) t \rightarrow s[t/x] \\
 (\text{case}) \quad & \text{case}_{\text{Typ}} (c \ s_1 \ \dots \ s_{\text{ar}(c)}) \text{ of } \{ \dots; (c \ x_1 \ \dots \ x_{\text{ar}(c)}) \rightarrow t; \dots \} \\
 & \rightarrow t[s_1/x_1, \dots, s_{\text{ar}(c)}/x_{\text{ar}(c)}]
 \end{aligned}$$

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

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 $[\cdot]$

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

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

Call-by-Name-Reduktion

Definition

Reduktionskontexte R in KFPT werden durch die folgende Grammatik mit Startsymbol RCtxt erzeugt:

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

Call-by-Name-Reduktion (2)

Definition

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

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

Beispiele

$$\begin{aligned}(\lambda x.x) ((\lambda y.y) (\lambda z.z)) &\rightarrow x[(\lambda y.y) (\lambda z.z)/x] \\ &= (\lambda y.y) (\lambda z.z)\end{aligned}$$

ist eine Call-by-Name-Reduktion.

$$\begin{aligned}(\lambda x.x) ((\lambda y.y) (\lambda z.z)) &\rightarrow (\lambda x.x) (y[(\lambda z.z)/y]) \\ &= (\lambda x.x) (\lambda z.z)\end{aligned}$$

ist **keine** Call-by-Name-Reduktion

Normalformen

Ein KFPT-Ausdruck s ist eine

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

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

Terminierung bzw. Konvergenz

Definition

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

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

Falls s nicht konvergiert, so sagen wir s **divergiert** und notieren dies mit $s\uparrow$.

Sprechweisen:

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

Dynamische Typisierung

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

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

Definition (Dynamische Typregeln für KFPT)

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

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

s ist **dynamisch ungetypt**

\iff

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

Beispiele

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

Dynamische Typisierung (2)

Satz

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

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

Die folgende Eigenschaft wird auch **Progress-Lemma** genannt:

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

Markierungsalgorithmus zur Call-by-Name-Redex-Suche

Für Ausdruck s starte mit s^* , Verschieberegeln:

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

Fälle danach:

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

Beispiel

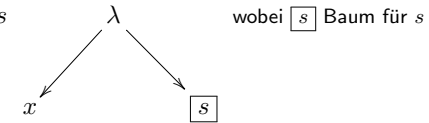
$$\begin{aligned}
 & (((\lambda x. \lambda y. (\text{caseList } y \text{ of } \{ \\
 & \quad \text{Nil} \rightarrow \text{Nil}; \\
 & \quad (\text{Cons } z \text{ } z s) \rightarrow (x z) \}) \text{ True})) (\lambda u, v. v)) (\text{Cons } (\lambda w. w) \text{ Nil}))^* \\
 \xrightarrow{\text{name}, \beta} & ((\lambda y. (\text{caseList } y \text{ of } \{ \\
 & \quad \text{Nil} \rightarrow \text{Nil}; \\
 & \quad (\text{Cons } z \text{ } z s) \rightarrow ((\lambda u, v. v) z) \}) \text{ True})) (\text{Cons } (\lambda w. w) \text{ Nil}))^* \\
 \xrightarrow{\text{name}, \beta} & (\text{caseList } (\text{Cons } (\lambda w. w) \text{ Nil}) \text{ of } \{ \\
 & \quad \text{Nil} \rightarrow \text{Nil}; \\
 & \quad (\text{Cons } z \text{ } z s) \rightarrow ((\lambda u, v. v) z) \}) \text{ True})^* \\
 \xrightarrow{\text{name}, \text{case}} & (((\lambda u, v. v) (\lambda w. w)) \text{ True})^* \\
 \xrightarrow{\text{name}, \beta} & ((\lambda v. v) \text{ True})^* \\
 \xrightarrow{\text{name}, \beta} & \text{True}
 \end{aligned}$$

Darstellung von Ausdrücken als Termgraphen

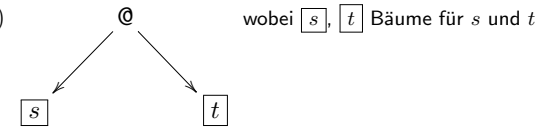
Knoten für je ein syntaktisches Konstrukt des Ausdrucks

- Variablen = ein Blatt

- Abstraktionen $\lambda x. s$



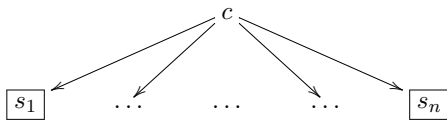
- Applikationen $(s t)$



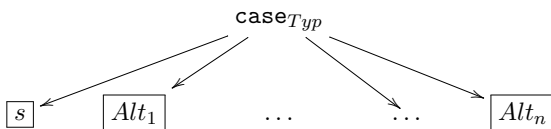
Darstellung von Ausdrücken als Termgraphen (2)

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

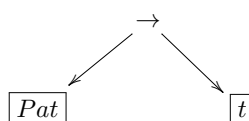
wobei s_i die Bäume für s_i



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

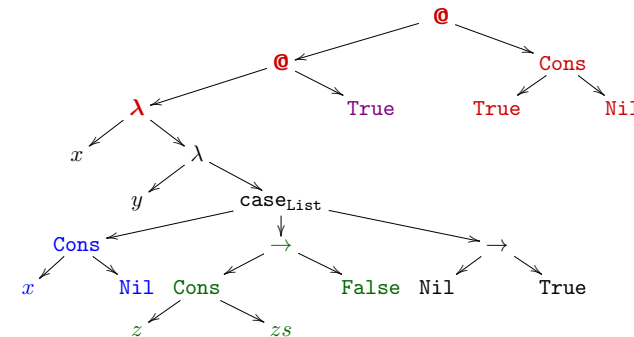


- case-Alternative $Pat \rightarrow t$



Beispiel

$$\left(\left(\lambda x. \lambda y. \text{caseList } (\text{Cons } x \text{ Nil}) \text{ of } \{ \right. \right. \\
 \quad (\text{Cons } z \text{ } z s) \rightarrow \text{False}; \\
 \quad \text{Nil} \rightarrow \text{True} \left. \left. \right) \text{ True} \right) (\text{Cons } \text{True Nil})$$



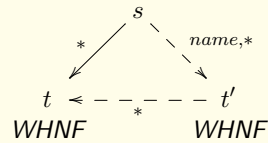
CBN-Redex-Suche: immer links, bis Abstraktion, Konstruktorenanwendung

Call-by-Name-Reduktion: Eigenschaften

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

Theorem (Standardisierung für KFPT)

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



Die Kernsprache KFPTS

Rekursive Superkombinatoren: KFPTS

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

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

Beispiel: Superkombinator length

```
length xs = caseList xs of {
  Nil → 0;
  (Cons y ys) → (1 + length ys)}
```

KFPTS: Syntax

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

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

KFPTS: Syntax (2)

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

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

dabei

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

Beispiel: Definition des Superkombinators map :

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

KFPTS: Syntax (2) Superkombinatoren

Haskell: Unterschiede bei Superkombinatordefinitionen:

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

KFPTS: Syntax (3)

Ein **KFPTS-Programm** besteht aus:

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

Dabei müssen alle in s verwendeten Superkombinatoren auch definiert sein.

KFPTS: Operationale Semantik

Reduktionskontexte:

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

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

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

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

$$(\mathbf{SK}-\beta) \quad (\mathbf{SK} \ s_1 \ \dots \ s_n) \rightarrow e[s_1/x_1, \dots, s_n/x_n], \\ \text{wenn } \mathbf{SK} \ x_1 \ \dots \ x_n = e \text{ die Definition von } \mathbf{SK} \text{ ist}$$

Call-by-Name-Reduktion:

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

KFPTS: WHNFs und Dynamische Typisierung

WHNFs

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

Direkt dynamisch ungetypt:

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

Markierungsalgorithmus

Markierung funktioniert genauso wie in KFPTS:

- $(s\ t)^* \Rightarrow (s^*\ t)$
- $(case_{Typ}\ s\ of\ Alts)^* \Rightarrow (case_{Typ}\ s^*\ of\ Alts)$

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 ($case\ [.] \dots$): direkt dynamisch ungetypt

Beispiel

Die Superkombinatoren map und not seien definiert als:

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

Beispiel zur Auswertung:

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

WHNF erreicht!

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

Erweiterung um seq

Erweiterung um seq

seq ist in KFPT, KFPTS nicht kodierbar!

Wir bezeichnen mit

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

Wir verzichten auf die formale Definition!

Man benötigt u.a. die Reduktionsregel:

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

erweiterte Reduktionskontexte und die neue Verschieberegeln:

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

Erweiterung um polymorphe Typen: KFPTSP

KFPTSP

Mit KFPTSP bezeichnen wir **polymorph getyptes** KFPTS

Definition

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

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

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

Beispiele

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

Einfache Typregeln

- Für die Anwendung:

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

- Instantiierung

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

- 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

and := $\lambda x, y. \text{case}_{\text{Bool}}\ x\ \text{of } \{\text{True} \rightarrow y; \text{False} \rightarrow \text{False}\}$
 or := $\lambda x, y. \text{case}_{\text{Bool}}\ x\ \text{of } \{\text{True} \rightarrow \text{True}; \text{False} \rightarrow y\}$

Mit der Anwendungsregel:

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

Beispiel

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

Beispiel

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

KFPTSP+seq als Kernsprache für Haskell

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

Zahlen

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

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

$$\mathcal{P}(0) := \text{Zero}$$
$$\mathcal{P}(n) := \text{Succ}(\mathcal{P}(n-1)) \text{ für } n > 0$$

Z.B. 3 als `Succ(Succ(Succ(Zero)))`.

let und where

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

Funktionen, für Peano-Zahlen

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

Funktionen, für Peano-Zahlen (2)

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

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

if-then-else und case

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

case-Ausdrücke: Beispiel

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

wird zu (noch in Haskell-Syntax):

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

Funktionsdefinitionen

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

Funktionen: Beispiele

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

wird zu:

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

Funktionen: Beispiele (2)

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

wird zu:

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

Übersicht

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

Lazy Evaluation in Haskell

Lazy Evaluation in Haskell

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

Ideen:

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

Beispiel: Als Darstellung mit Graphen

Eine alternative Darstellung ist die Verwendung von Graphen:



Beispiel

Betrachte `square (10+5)` wobei

`square x = x*x`

- Call-by-Name-Auswertung:
 $\text{square } (10+5) \xrightarrow{\text{name}} (10+5) * (10+5) \xrightarrow{\text{name}} 15 * (10+5) \xrightarrow{\text{name}} 15 * 15 \xrightarrow{\text{name}} 225.$
- Call-by-Value-Auswertung:
 $\text{square } (10+5) \xrightarrow{\text{value}} \text{square } 15 \xrightarrow{\text{value}} 15 * 15 \xrightarrow{\text{value}} 225.$
- Call-by-Need-Auswertung (`<i>`) repräsentieren Referenzen
 $(\text{square } \langle 1 \rangle, \{ \langle 1 \rangle \mapsto 10+5 \})$
 $\xrightarrow{\text{need}} (\langle 1 \rangle * \langle 1 \rangle, \{ \langle 1 \rangle \mapsto 10+5 \})$
 $\xrightarrow{\text{need}} (15 * 15, \{ \langle 1 \rangle \mapsto 15 \})$
 $\xrightarrow{\text{need}} (225, \{ \langle 1 \rangle \mapsto 15 \}).$

Beispiel mit Trace

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

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

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

Auswertung von `z`

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

Beispiel mit Divergenz

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

Auswertung einer Anwendung `foo e1 e2 e3`:

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

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

Noch ein Beispiel

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

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

Potentiell unendliche Datenstrukturen

Lazy Evaluation ermöglicht „unendliche“ Datenstrukturen:

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

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

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

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

```
nums = iterate (1+) 0
```

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

Thunks im GHCi

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

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


Thunks im GHCi

Mit `:print`:

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

Thunks im GHCi (3)

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

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

Thunks im GHCi (4)

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

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

Trennung von Daten und Kontrollfluss

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

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

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

Trennung von Daten und Kontrollfluss (2)

Weiteres Beispiel: Sieb des Erathostenes

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

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

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

Zirkularität

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

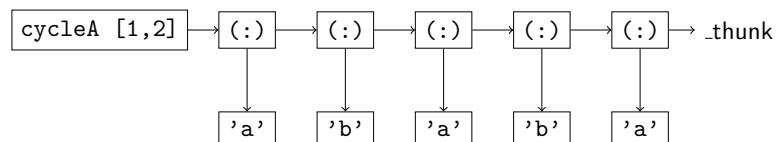
```
cycleA xs = xs ++ cycleA xs
```

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

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

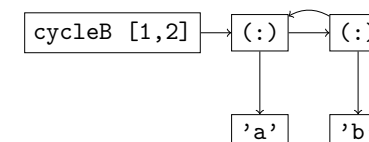
Zirkularität (2)

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



Zirkularität (3)

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



Strikte Auswertung

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

Spracherweiterung BangPatterns:

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

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

Strikte Datentypen (2)

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

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

Strikte Datentypen

Der GHC erlaubt auch die Deklaration von strikten Datentypen:

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

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

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

Lazy Pattern

Lazy Pattern-Matches: Versehe das Pattern mit ~:

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

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

Falsch:

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

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

Lazy-Pattern (2)

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

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

wird

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

was wiederum übersetzt wird in

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

oder kürzer geschrieben:

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

Beispiel

Funktion `splitAt`:

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

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

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

sofort 'a' zurück.

Bei Verwendung eines normalen Pattern Matches, dauert dies viel länger (da erst 10 Millionen rekursive Aufrufe berechnet werden)