Kapitel 5: λ-Kalkül und Laziness

Lernziele dieses Kapitels

1. Syntax des einfachen, ungetypten λ-Kalküls
   - Symbolische Manipulation von λ-Ausdrücken in Haskell, Substitution
2. Reduktion
   - Reduktionsarten, Berechnungsfortschritt
   - Strategien und Normalformen, Fixpunktkombinator, Church-Rosser-Theorem
   - Lazy/Eager-Auswertung; explizite Nachbildung im Programm
3. Kodierung von Datentypen im ungetypten λ-Kalkül
4. Alternativen zur λ-Abstraktion / zu Variablennamen
5. Einfach getypter λ-Kalkül
   - Funktionen höherer Ordnung, Currying

λ-Kalkül

- Modell für Berechenbarkeit
  - Äquivalenz zu Turing-Maschinen
- Formalisierung des Funktionenbegriffs
  - intensional: konstruktiv, durch Berechnungsvorschrift
  - symbolisch manipulierbar: Komposition, Applikation etc.
- Grundlage der funktionalen Programmiersprachen
- ungetypter Kalkül mit minimaler Syntax
  - nur drei Konstrukte: Variable, Applikation und Lambda-Abstraktion
  - Funktionen mit einem Argument reichen aus (Currying)

Syntax des minimalen, ungetypten λ-Kalküls

Sei V eine abzählbare Menge von Variablen. Die Sprache Λ der λ-Terme ist induktiv definiert über dem Alphabet $V \cup \{(,), \lambda\}$:

(i)	$x \in V$	$\Longrightarrow$	$x \in \Lambda$	Variable
(ii)	$M, N \in \Lambda$	$\Longrightarrow$	$(MN) \in \Lambda$	Applikation
(iii)	$M \in \Lambda \ \wedge \ x \in V$	$\Longrightarrow$	$(\lambda x M) \in \Lambda$	Lambda-Abstraktion

Abkürzende Schreibweise (mit "$\bullet$" hinter den Parametern):

$$\lambda x_1 \ldots x_n \bullet M = (\lambda x_1 \ldots (\lambda x_n M) \ldots)$$

$$M N_1 N_2 \ldots N_n = (\ldots ((M N_1) N_2) \ldots N_n)$$

Λ als algebraischer Datentyp

```
data LExp = V String          Variable
          | LExp :@: LExp     Applikation
          | L String LExp     Lambda-Abstraktion
          deriving (Show)

fv, bv :: LExp -> [String]

fv (V x)      = [x]                          freie Variablen
fv (f :@: x) = union (fv f) (fv x)
fv (L x e)    = fv e \\ [x]

bv (V _)      = []                           gebundene Variablen
bv (f :@: x) = union (bv f) (bv x)
bv (L x e)    = union (bv e) [x]
```

Substitution

- Notation: $E\,[x := A]$
- Wirkung: ersetze in E jedes freie Vorkommen von x durch A
- gebundene Namen in E, die frei in A vorkommen, müssen umbenannt werden!

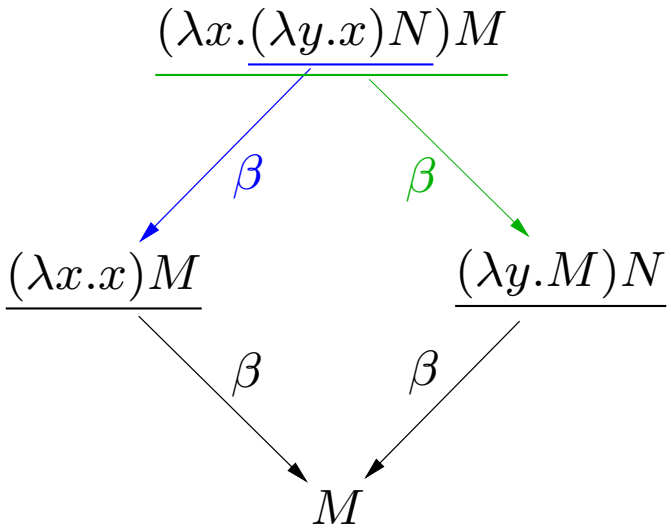
```
substitute :: String -> LExp -> LExp -> LExp
substitute x a = s where
 s (V w) | x==w = a
         | x/=w = V w
 s (f :@: x)    = s f :@: s x
 s (L w exp)
  | x==w = L w exp
  | x/=w = let fresh = occursNot ([w,x] ++ fv exp ++ fv a)
           in L fresh (s (substitute w (V fresh) exp))
```

Reduktions- und Konversionsarten

- β-Reduktion: $(\lambda x.E)\ A \rightarrow_\beta E\,[x := A]$
  - formale Beschreibung der Funktionsapplikation
  - Haskell-Bsp.: `(\x -> x*x) 5` $\rightarrow_\beta$ `5*5`
- δ-Reduktion: Bsp.: `5*5` $\rightarrow_\delta$ `25`
  - Applikation einer vordefinierten Funktion (nicht im minimalen λ-Kalkül)
- α-Konversion: $y \notin \mathrm{fv}(E) \implies (\lambda x.E) =_\alpha (\lambda y.E[x := y])$
  - Umbenennung einer gebundenen Variable (für Substitution)
- η-Konversion: $x \notin \mathrm{fv}(E) \implies (\lambda x.E\,x) =_\eta E$
  - Wechsel von/zur extensionalen Form

Redex, Reduktionsreihenfolge, Normalform

- Redex: Reducible expression
- Reihenfolge der Reduktionen beeinflusst das Ergebnis nicht (Leibniz-Regel)
  $\Rightarrow$ lokale Konfluenz der β-Reduktion



- Ausdruck ohne Redex: Normalform
- leftmost-outermost Redex: beginnt am weitesten links
- leftmost-innermost Redex: endet am weitesten links

λ-Terme ohne Normalform

1.
   - $\Omega \stackrel{def}{=} (\lambda x.(x\,x))\,(\lambda x.(x\,x))$.
   - Es gibt nur einen Redex, den gesamten Ausdruck.
   - $\Omega \rightarrow_\beta \Omega \rightarrow_\beta \Omega \rightarrow_\beta \ldots$
2.
   - Fixpunktkombinator Y, angewendet auf Variable (f):
   - $Y \stackrel{def}{=} \lambda h.(\lambda x.h\,(x\,x))\,(\lambda x.h\,(x\,x))$
   - $Y\,f = (\lambda h.(\lambda x.h\,(x\,x))\,(\lambda x.h\,(x\,x)))\,f$
     $\rightarrow_\beta (\lambda x.f\,(x\,x))\,(\lambda x.f\,(x\,x))$
     $\rightarrow_\beta f\,((\lambda x.f\,(x\,x))\,(\lambda x.f\,(x\,x)))$
     $= f\,(Y\,f)$
   - Anwendung: Rekursion
     Rekursion terminiert, wenn $Y\,f$ von f eliminiert wird

Rekursion am Beispiel der Fakultätsfunktion

Arithmetik und if speziell durch λ-Terme kodiert (später)

$$\text{fac} \stackrel{def}{=} \lambda\, h\, n.\ \text{if}\ n{=}0\ \text{then}\ 1\ \text{else}\ n * h\,(n{-}1)$$

$$\begin{aligned}
&Y\ \text{fac}\ 3 \rightarrow_\beta\ \text{fac}\ (Y\ \text{fac})\ 3\\
&\rightarrow_\beta\ (\lambda n.\ \text{if}\ n{=}0\ \text{then}\ 1\ \text{else}\ n * Y\ \text{fac}\ (n{-}1))\ 3\\
&\twoheadrightarrow_\beta\ 3 * Y\ \text{fac}\ 2 \qquad [\twoheadrightarrow_\beta\text{: Folge von } \rightarrow_\beta]\\
&\rightarrow_\beta\ 3 * \text{fac}\ (Y\ \text{fac})\ 2\\
&\rightarrow_\beta\ 3 * (\lambda n.\ \text{if}\ n{=}0\ \text{then}\ 1\ \text{else}\ n * Y\ \text{fac}\ (n{-}1))\ 2\\
&\twoheadrightarrow_\beta\ 3 * (2 * Y\ \text{fac}\ 1)\\
&\rightarrow_\beta\ 3 * (2 * \text{fac}\ (Y\ \text{fac})\ 1)\\
&\rightarrow_\beta\ 3 * (2 * (\lambda n.\ \text{if}\ n{=}0\ \text{then}\ 1\ \text{else}\ n * Y\ \text{fac}\ (n{-}1))\ 1)\\
&\twoheadrightarrow_\beta\ 3 * (2 * (1 * Y\ \text{fac}\ 0))\\
&\rightarrow_\beta\ 3 * (2 * (1 * \text{fac}\ (Y\ \text{fac})\ 0))\\
&\rightarrow_\beta\ 3 * (2 * (1 * (\lambda n.\ \text{if}\ n{=}0\ \text{then}\ 1\ \text{else}\ n * Y\ \text{fac}\ (n{-}1))\ 0))\\
&\twoheadrightarrow_\beta\ 3 * (2 * (1 * 1)) \twoheadrightarrow_\beta\ 6
\end{aligned}$$

Church-Rosser Theorem und Folgerungen

Definitionen

- $\twoheadrightarrow_\beta$: reflexiv-transitive Hülle von $\rightarrow_\beta$
- $=_\beta$: reflexiv-symmetrisch-transitive Hülle von $\rightarrow_\beta$

Church-Rosser Theorem (lokale Konfluenz von $\rightarrow_\beta$ berücksichtigt (Folie 5/7))

$$M =_\beta N \implies \exists P : M \twoheadrightarrow_\beta P \wedge N \twoheadrightarrow_\beta P$$

Folgerungen

1. Falls M eine (β-)Normalform N hat, dann gilt: $M \twoheadrightarrow_\beta N$ (Existenz einer Folge von β-Reduktionen von M nach N)
2. Ein λ-Term hat höchstens eine (β-)Normalform (Eindeutigkeit bis auf die anderen Reduktionsarten, insbesondere Umbenennung gebundener Variablen)

Reduktionsstrategien (1)

Bsp.: $\text{fac}\ (Y\ \text{fac})\ 2$

1. normal-order Reduktion
   - Wahl des leftmost-outermost Redex (Anfang am weitesten links)
   - $(\ (\ \underline{\text{fac}\ (Y\ \text{fac})}\ )\ 2\ )$
     $\rightarrow_\beta\ (\lambda n.\ \text{if}\ n{=}0\ \text{then}\ 1\ \text{else}\ n\ *\ Y\ \text{fac}\ (n{-}1))\ 2$

2. applicative-order Reduktion
   - Wahl des leftmost-innermost Redex (Ende am weitesten links)
   - $(\ (\ \text{fac}\ (\ \underline{Y\ \text{fac}}\ )\ )\ 2\ )$
     $\rightarrow_\beta\ \ \text{fac}\ (\text{fac}\ (Y\ \text{fac}))\ 2$

Reduktionsstrategien (2)

Satz: Wenn ein λ-Term eine Normalform besitzt, dann wird diese immer durch normal-order Reduktion erreicht.

Anm.: applicative-order Reduktion hat diese Eigenschaft nicht

Bsp.:

- normal order: $\underline{(\lambda\, x\, y\, .\, y)\ (\ (\lambda x.(x\, x))\, (\lambda x.(x\, x))\ )} \rightarrow_\beta (\lambda\, y.\, y)$
- applicative order: $(\lambda\, x\, y\, .\, y)\ (\ \underline{(\lambda x.(x\, x))\, (\lambda x.(x\, x))}\ )$
  $\rightarrow_\beta (\lambda\, x\, y\, .\, y)\ (\ (\lambda x.(x\, x))\, (\lambda x.(x\, x))\ )$

Reduktionsstrategien (3)

lazy-Auswertung von Haskell:

- normal order Reduktion
- Unikate für gemeinsame Teilausdrücke (Sharing)
- nicht-strikte Konstruktorapplikation (keine vorzeitige Ausw.)

Strategie:	applicative order	normal order	lazy evaluation
Applikation	strikt	nicht-strikt	
Parameterauswertung	bei Reduktion	bei Benutzung	bei erster Benutzung
Probleme	Nichttermination	Komplexität	Speicherplatz
Programmiersprachen	ML, OCaml	—	Haskell
Parameterübergabe	call-by-value	call-by-name	call-by-need

Eingeschränkte Normalformen (1)

- weak normal form (WNF), alle Redexe sind innerhalb von λ-Abstraktionen
- weak head normal form (WHNF), kein Redex, der am Anfang des λ-Ausdrucks beginnt

Beispiele im minimalen λ-Kalkül:

$(\lambda x.y)$	in Normalform, kein Redex
$(\lambda x.\underline{(\lambda y.y)\ z})$	nicht in Normalform, aber in WNF
$x\ \underline{((\lambda y.y)\ z)}$	nicht in WNF, aber in WHNF
$\underline{(\lambda x.x)\ ((\lambda y.y)\ z)}$	nicht in WHNF

Eingeschränkte Normalformen (2)

- weak normal form (WNF), schwächer als Normalform
  - Form: alle Redexe sind innerhalb von λ-Abstraktionen
  - Programmiersprachen: OCaml, ML
  - Bedeutung: keine "Optimierung" von Funktionen zur Laufzeit
  - OCaml-Beispiele (`(fun x -> f)` für $(\lambda x.f)$)
    - nicht WNF: Ausdruck mit Redex nicht innerhalb einer λ-Abstraktion

```
# let x = (fun y -> 1/y) 0;;
Exception: Division_by_zero.
```

    - WNF: Ausdruck mit Redex nur innerhalb einer λ-Abstraktion

```
# let f = (fun x -> (fun y -> 1/y) 0);;
val f : 'a -> int = <fun>
# let f x = (fun y -> 1/y) 0;; dasselbe mit syntactic sugar
val f : 'a -> int = <fun>
```

Eingeschränkte Normalformen (3)

- weak head-normal form (WHNF), schwächer als WNF
  - Form: kein Redex beginnt am Anfang des gesamten λ-Ausdrucks
  - Programmiersprache: Haskell
  - Bedeutung: keine Auswertung von Konstruktorargumenten

    * OCaml (Bsp. in WHNF, aber OCaml wertet aus bis WNF)

      ```
      # let xs = [ 1, 2, 3/0 ];;          Redex
      Exception: Division_by_zero.
      ```

    * Haskell (wertet nur aus bis WHNF)

      ```
      Prelude> let xs = [ 1, 2, 3/0 ]     kein Redex am Anfang
      Prelude> (kein Fehler)
      ```

Eager-Auswertung in Haskell (1)

- f `$!` x: Auswertung von x nach WHNF vor Anwendung der Funktion f

```
Prelude> (const "foo") $ (error "bar")
"foo"
Prelude> (const "foo") $! (error "bar")     Auswertung
"*** Exception: bar
Prelude> (const "foo") $! [error "bar"]     [.] ist in WHNF
"foo"
```

- x `` `seq` `` y: Auswertung von x nach WHNF, dann Rückgabe von y

```
Prelude> let x = error "bar"  in  x `seq` 5
*** Exception: bar
Prelude> let x = error "bar"  in  5
5
```

Eager-Auswertung in Haskell (2)

Laziness bewirkt: Speicher oft gefüllt mit unerledigter Arbeit
Striktheits-Annotationen für

- Reduktion des Speicherverbrauchs
- kurze Antwortzeit bei Suche (Datenstruktur muss nicht erst erzeugt werden)
- Glasgow Parallel Haskell: parallele Prozesse sollen sofort mit der Arbeit beginnen

Problem bei Datenstrukturen: Auswertung nach WHNF reicht nicht
Lösungen zum Erreichen von hyperstrictness:

1. Striktheitsannotationen an allen Funktionen, die die Struktur erzeugen
2. eleganter: Striktheitsannotation der Argumente der Datenkonstruktoren

```
data Tree a = Leaf !a | Fork !a !(Tree a) !(Tree a)
```

Laziness in OCaml

Ausnutzung der WNF (keine Auswertung innerhalb von λ-Abstraktionen).

- unendliche Listen (Produzenten: delay, Konsumenten: force)

```
type 'a lazylist = Nil | Cons of 'a * (unit -> 'a lazylist)
let rec from n = Cons (n, fun _ -> from (n+1))
let head (Cons (x,_)) = x
let tail (Cons (_,xs)) = xs
let rec take n xs = if n=0 || xs=Nil
                    then []
                    else head xs :: take (n-1) ((tail xs) ())
```

- Verwendung

```
# from 4;;
- : int lazylist = Cons (4, <fun>)
# take 10 (from 4);;
- : int list = [4; 5; 6; 7; 8; 9; 10; 11; 12; 13]
```

Kodierung boolescher Ausdrücke

Bedingung an `True`, `False` und `If`:

$(\texttt{If True}\ M\ N\ =\ M)\ \wedge\ (\texttt{If False}\ M\ N\ =\ N)$

Lösung: $\{\ \texttt{True} \equiv \lambda x\, y.x,\quad \texttt{False} \equiv \lambda x\, y.y,\quad \texttt{If} \equiv \lambda\, i\ t\ e\ .\ i\ t\ e\ \}$

Beweis:

- $\texttt{If True}\ M\ N$
  $(\lambda\, i\ t\ e\ .\ i\ t\ e)\ (\lambda x\, y.x)\ M\ N \twoheadrightarrow_\beta\ (\lambda x\, y.x)\ M\ N \rightarrow_\beta\ (\lambda y.M)\ N \rightarrow_\beta\ M$
- $\texttt{If False}\ M\ N$
  $(\lambda\, i\ t\ e\ .\ i\ t\ e)\ (\lambda x\, y.y)\ M\ N \twoheadrightarrow_\beta\ (\lambda x\, y.y)\ M\ N \rightarrow_\beta\ (\lambda y.y)\ N \rightarrow_\beta\ N$

Kodierung natürlicher Zahlen

Church-Numerale: $\boxed{\lceil n \rceil \equiv \lambda f x.f^n\, x}$
(f^n die n-fache Applikation von f)

Zähloperationen:

- $\texttt{Succ}\lceil n \rceil = \lceil n{+}1 \rceil$:
  $\texttt{Succ} \equiv \lambda n\, f\, x.f\, (n\, f\, x)$
- $\texttt{Pred}\lceil n \rceil = \lceil \texttt{max}\, (n{-}1)\, 0 \rceil$:
  $\texttt{Pred} \equiv \lambda n\, f\, x.n\, (\lambda y\, z.z\, (y\, f))\, ((\lambda x\, y.x)\, x)\, (\lambda x.x)$
- $\texttt{IsZero}\lceil 0 \rceil = \texttt{True}$; $n{>}0 \implies \texttt{IsZero}\lceil n \rceil = \texttt{False}$
  $\texttt{IsZero} \equiv \lambda n.n\, (\lambda x.\texttt{False})\, \texttt{True}$

Anmerkung: eine 2-Zähler-Maschine kann eine beliebige Turing-Maschine simulieren (Hopcroft/Ullman: Einführung in die Automatentheorie, Satz 7.9)

Kodierung arithmetischer Operationen

Idee: betrachte $\lceil n \rceil\ f\ x$ als n-fache Anwendung von f auf x

vgl. `multiple(n) f x = foldr (const f) x [1..n]`

$\lceil n+m \rceil$: `Add` $\equiv \lambda n\, m\, f\, x.\ n\, f\, (m\, f\, x)$

`multiple(3) succ (multiple(5) succ 0)` $\rightsquigarrow$ `8`

$\lceil n*m \rceil$: `Mult` $\equiv \lambda n\, m\, f.\ n\, (m\, f)$

`multiple(3) (multiple(5) succ) 0` $\rightsquigarrow$ `15`

$\lceil n^m \rceil$: `Exp` $\equiv \lambda n\, m\, f\, x.\ m\, n\, f\, x$

`multiple(3) (multiple(5)) succ 0` $\rightsquigarrow$ `125`

$\lceil n \dot{-} m \rceil$: `Sub` $\equiv \lambda n\, m\, .\ m$ `Pred` n

`multiple(3) pred (multiple(5) succ 0)` $\rightsquigarrow$ `2`

m: Zahl arbeitet als Iterator, n: Zahl selbst

Applikation ist nicht assoziativ

Gegenbeispiel:

```
*Multiple> let m3 = multiple 3
*Multiple> (m3 (m3 succ)) 0   -- berechnet (*3).(*3)
9
*Multiple> ((m3 m3) succ) 0   -- berechnet (*3).(*3).(*3)
27
```

Selbstapplikation `m3 m3` ?

```
*Multiple> :t m3
m3 :: forall a. (a -> a) -> a -> a
m3 :: ((Int->Int)->(Int->Int)) -> (Int->Int) -> (Int->Int)
m3 :: (Int->Int)->(Int->Int)
```

Polymorphie erlaubt Verwendung von `m3` mit mehreren Typen!

Monomorphie-Restriktion

Haskell unterstützt λ-Abstraktionen:

```
(*Multiple> (\ m3 -> m3 (m3 succ) 0) (multiple 3)
9
*Multiple> (\ m3 -> m3 m3 succ 0) (multiple 3)
<interactive>:1:
  Occurs check: cannot construct the infinite type: t = t -> t1 -> t2
        Expected type: t
        Inferred type: t -> t1 -> t2
    In the first argument of `m3', namely `m3'
    In a lambda abstraction: \ m3 -> (m3 m3 succ) 0
```

Monomorphie-Restriktion (aus Effizienzgründen)

`let`-Definitionen haben diese Beschränkung nicht:

```
*Multiple> let m3 = multiple 3  in  m3 m3 succ 0
27
```


Kodierung von Listen (Kellern)

$$\texttt{Cons} \equiv \lambda x\, y\, f.f\, x\, y$$

$$\texttt{Head} \equiv \lambda c.c\,(\lambda a\, b.a)$$

$$\texttt{Tail} \equiv \lambda c.c\,(\lambda a\, b.b)$$

$$\begin{aligned}
& \binom{\texttt{Head}}{\texttt{Tail}}\ (\texttt{Cons}\ h\ t) \\
\equiv\ & (\lambda c.c\,(\lambda a\, b.\tbinom{a}{b}))\ ((\lambda x\, y\, f.f\, x\, y)\ h\ t) \\
\rightarrow_\beta\ & (\lambda c.c\,(\lambda a\, b.\tbinom{a}{b}))\ (\lambda f.f\, h\, t) \\
\rightarrow_\beta\ & (\lambda f.f\, h\, t)\,(\lambda a\, b.\tbinom{a}{b}) \\
\rightarrow_\beta\ & (\lambda a\, b.\tbinom{a}{b})\, h\, t \\
\rightarrow_\beta\ & \tbinom{h}{t}
\end{aligned}$$

Hopcroft/Ullman: Einführung in die Automatentheorie, Lemma 7.3:
Eine beliebige einbändige TM kann durch eine det. 2-Kellermaschine simuliert werden. (Aufteilung des Bandes an der Kopfposition.)

de-Bruijn-Indizes

Variablen-Umbenennung ist vermeidbar!

- ersetze jedes Vorkommen einer λ-gebundenen Variablen durch $\mathtt{L}\ i$, wobei i die relative Schachtelungstiefe zur entsprechenden λ-Abstraktion ist
- lösche alle Variablen hinter den λ's
- Beispiel: $\lambda x.\lambda y.\lambda f.f\ (\lambda x.x)\ (x+y)$
  in der de-Bruijn-Form: $\lambda.\lambda.\lambda.(\mathtt{L}\,0)\ (\lambda.\mathtt{L}\,0)\ (\mathtt{L}\,2+\mathtt{L}\,1)$
- Vorteil: einfach zu implementieren (categorical abstract machine)
- Nachteil: schwer manuell benutzbar

SK-Kombinator-Reduktion

λ-Abstraktionen können ersetzt werden!

Transformationsregeln in die SK-Form:

$$
\begin{array}{lcl}
\lambda x.x & \longrightarrow & I \longrightarrow S\,K\,K \\
\lambda x.c & \stackrel{c \neq x}{\longrightarrow} & K\,c \\
\lambda x.e_1\,e_2 & \longrightarrow & S\,(\lambda x.e_1)\,(\lambda x.e_2)
\end{array}
$$

Reduktionsregeln:

$$
\begin{array}{lcl}
S\;f\;g\;x & \rightarrow & f\;x\;(g\;x) \\
K\;c\;x & \rightarrow & c
\end{array}
$$

Implementierung: durch Graphreduktion (Termersetzung mit Sharing)

Typen im λ-Kalkül

Vorteile der Verwendung eines statischen Typsystems:

1. Erhöhung der Sicherheit des Programmentwurfs
   - viele logische Fehler äussern sich als Typfehler
   - automatisch überprüfbare Dokumentation des Programms
2. Effizienzsteigerung durch
   - Vermeidung von Typüberprüfungen zur Laufzeit
   - Speicherersparnis: Vermeidung von Information über Typ oder Speicherplatzbedarf in den Datenobjekten
   - Anwendbarkeit *sicherer* Programmoptimierungen
     (`(2*x+y)-x` → `x+y` richtig für `Integer`, falsch für `Float`)

Der einfach getypte λ-Kalkül

- jedem Ausdruck wird ein Typ zugewiesen
- nur *ein* Typkonstruktor: $\rightarrow$ (Infix, rechtsassoziativ)
- Grundttyp: $0 \in Typ$ (später Typvariablen)
- $\sigma, \tau \in Typ \implies (\sigma \rightarrow \tau) \in Typ$

Variable:	$x \in V^{\sigma}$	$\implies$	$x \in \Lambda_{\sigma}$
Applikation:	$M \in \Lambda_{\sigma \rightarrow \tau} \ \wedge \ N \in \Lambda_{\sigma}$	$\implies$	$(MN) \in \Lambda_{\tau}$
Abstraktion:	$M \in \Lambda_{\tau} \ \wedge \ x \in \Lambda_{\sigma}$	$\implies$	$(\lambda x.M) \in \Lambda_{\sigma \rightarrow \tau}$

Der Typ des Fixpunktoperators Y (1)

Der λ-Ausdruck für Y ist (monomorph) in Haskell nicht typisierbar:

```
Prelude> :t (\ h -> (\ x -> h (x x)) (\ x -> h (x x)))
<interactive>:1:
    Occurs check: cannot construct the
                  infinite type: t = t -> t1
```

Typinferenzregel im einfach getypten λ-Kalkül (keine Polymorphie):

$$\frac{\mathtt{x} :: 0 \mid \mathtt{x} :: 0 \to 0}{\mathtt{x}\ \mathtt{x} :: 0}$$

Die durch `x` induzierte Gleichung: $0 = 0 \to 0$ ist nicht erfüllbar!

Der Typ des Fixpunktoperators Y (2)

auch Ersetzung von λ durch polymorphes `let` in Haskell reicht nicht:

```
y h = let f1 x = h (x x)
          f2 x = h (x x)
      in f1 f2
```

Fehlermeldung:

```
Occurs check: cannot construct the infinite type: t = t -> t1
    Expected type: t
    Inferred type: t -> t1
In the first argument of ‘x’, namely ‘x’
In the first argument of ‘h’, namely ‘(x x)’
```

Der Typ des Fixpunktoperators Y (3)

rekursive Definition in Haskell durch Angabe der Gleichung:

```
Prelude> let y h = h (y h)
Prelude> :t y
forall t. (t -> t) -> t
```

Anm.: Rekursion gibt es im einfach getypten λ-Kalkül nicht!

mit der lazy-Auswertung von Haskell leistet `y` das Gewünschte.

Bsp.: Fakultätsfunktion via `y` und Fixpunktfunktional `fpfac`

```
Prelude> let fpfac f n = if n==0 then 1 else n * f (n-1)
Prelude> map (y fpfac) [0..8]
[1,1,2,6,24,120,720,5040,40320]
```

Sätze über den einfach getypten λ-Kalkül

1. Jede Reduktionssequenz terminiert.
2. Folgerung aus (1.): der einfach getypte λ-Kalkül ist nicht Turing-vollständig.

Ordnung ($\mathcal{O}$) und Kardinalität (#)

	Typ	Ordnung ($\mathcal{O}$) des Typs
`Bool`	$\mathbb{B}$	$\mathcal{O}\,\mathbb{B} = 0$
`[]`	$\alpha \to [\alpha]$	$\mathcal{O}\,[\alpha] = \mathcal{O}\,\alpha$
`(->)`	$\alpha \to \beta \to (\alpha \to \beta)$	$\mathcal{O}(\alpha \to \beta) = max\,\{\mathcal{O}\alpha + 1, \mathcal{O}\beta\}$
`(,)`	$\alpha \to \beta \to (\alpha, \beta)$	$\mathcal{O}(\alpha, \beta) = max\{\mathcal{O}\alpha, \mathcal{O}\beta\}$
`(,,)`	$\alpha \to \beta \to \gamma \to (\alpha, \beta, \gamma)$	$\mathcal{O}(\alpha, \beta, \gamma) = max\{\mathcal{O}\alpha, \mathcal{O}\beta, \mathcal{O}\gamma\}$

- $\mathcal{O}((\mathbb{B}, \mathbb{B}) \to \mathbb{B}) = 1$
  $\#((\alpha, \beta) \to \gamma) = \#\gamma^{\#(\alpha,\beta)} = \#\gamma^{\#\alpha * \#\beta}$
- $\mathcal{O}(\mathbb{B} \to \mathbb{B} \to \mathbb{B}) = 1$
  $\#(\alpha \to \beta \to \gamma) = \#(\beta \to \gamma)^{\#\alpha} = (\#\gamma^{\#\beta})^{\#\alpha} = \#\gamma^{\#\alpha * \#\beta}$
- $\mathcal{O}((\mathbb{B} \to \mathbb{B}) \to \mathbb{B}) = 2$
  $\#((\alpha \to \beta) \to \gamma) = \#\gamma^{\#(\alpha \to \beta)} = \#\gamma^{\#\beta^{\#\alpha}}$

Isomorphie von $((\alpha, \beta) \to \gamma)$ und $(\alpha \to \beta \to \gamma)$

- Die Kardinalität beider Mengen ist $\#\gamma^{\#\alpha * \#\beta}$

- Bijektion zwischen beiden Mengen:

```
curry :: ((α,β) → γ) → (α → β → γ)
curry f x y = f (x,y)

uncurry :: (α → β → γ) → ((α,β) → γ)
uncurry f (x,y) = f x y
```