# Paper: The Promises of Functional Programming

Von Konrad Honsen

**Was unterscheidet Funktionen im mathematischen Sinn von Funktionen im Sinn imperativer oder objektorientierter Programmiersprachen?**

Funktionen im mathematischen Sinn sind rechtseindeutige Relationen zwischen Argument- und Bildbereich:

Ist eine Funktion für ein Argument definiert, gibt es genau einen Bildwert, einen Funktionswert für dieses Argument.

Im programmiersprachlichen Jargon sind Funktionen im mathematischen Sinn seiteneffektfrei. Für Funktionen einer imperativen oder objektorientierten Programmiersprache gilt das aber nicht. Funktionen einer imperativen oder objektorientierten Programmiersprache sind Subroutinen, die einen Wert zurückliefern und dabei Seiteneffekte bewirken können.

Sie können also über das Liefern eines Werts hinaus etwas tun und bewirken (vgl. S. 87, linke Spalte, Anfang von Abschnitt “Basics”). Angewendet auf dasselbe Argument liefern sie deshalb nicht dasselbe Resultat.

Für Funktionen im mathematischen Sinn ist das nicht möglich

# Paper: Scientific Programming - The Promises of Typed, Pure, and Lazy Functional Programming: Part II

Von Konstantin Läufer, George K. Thiruvathukal

**Welche Vorteile gibt es für statische gegenüber dynamischerTypisierung in Programmiersprachen?**

Zur Laufzeit eines Programm einer statisch getypten Programmiersprache sind keinerlei Fehler oder Programmabstürze aufgrund inkompatibler Typung von Ausdrücken, Variablen, etc. möglich.

Eine sehr große Gruppe von Fehlermöglichkeiten kann deshalb bereits zur Übersetzungszeit eines Programms ausgeschlossen werden. Zur Laufzeit eines Programms sind deshalb auch keine Typuberprüfungen mehr nötig, wie das für Programme dynamisch getypter Programmiersprachen erforderlich ist. Die Programmausführung statisch typgeprüfter Programme ist dadurch performanter (S. 70, linke Spalte, Mitte).

Im Unterschied dazu können in Programmen dynamisch getypter Programmiersprachen, Typfehler zur Laufzeit auftreten und zu Programmabbrüchen führen. Im Fall sicherheitskritischer Anwendungen (Flugsicherungssoftware, Kraftwerkssteuerungen, etc.) ist das fatal. In der Praxis führt dies zu der Praxis, den Binärcode installierter Programme zur Fehlerbeseitigung direkt zu verändern (was seinerseits schwierig und fehlerträchtig ist).

In Programmen statisch getypter Programmiersprachen können solche Fehler schon zur Übersetzungszeit der Programme aufgedeckt und vor Auslieferung der Programme berichtigt werden (S. 70, linke Spalte, oben).

Ein oft ins Feld geführter Einwand, dass der Programmierer höheren Aufwand hat, indem er stets Typen deklarieren muss, ist für viele Sprachen, darunter auch Haskell, entkräftet, da Typen automatisch inferiert werden können und deshalb nicht notwendig vom Programmierer angegeben werden müssen (S. 70, linke Spalte, unten; S. 70, mittlere Spalte, unten, und fortfolgende).

**Listen in Haskell sind homogen, die Elemente von Listen sind alle von ein und demselben Typ. Wie kann man in Haskell dennoch quasi heterogene Listen implementieren?**

Listen über einem algebraischen Summentyp definieren, um quasi heterogene Listen zu definieren. (s.S. 73, linke Spalte, unten):

data IntOrString = AnInt Int | AString String

add [] = (0,"")

add (AnInt i : xs) = let (k,t) = add xs in (i + k, t)

add (AString s : xs) = let (k,t) = add xs in (k, s ++ t)

Prelude> add [AnInt 3, AString "adsf", AnInt 7, AString "quer"]

(10,"adsfquer")

Für ‘zweitypige’ Listen würde der vordefinierte polymorphe Typ Either a b eine andere Möglichkeit bieten, den Typ IntOrString zu definieren:

type IntOrString = Either Int String

add [] = (0,"")

add (Left i : xs) = let (k,t) = add xs in (i + k, t)

add (Right s : xs) = let (k,t) = add xs in (k, s ++ t)

Prelude> add [Left 3, Right "adsf", Left 7, Right "quer"]

(10,"adsfqwer")

**Was bedeutet der Begriff *referential transparency*?**

Referentielle Transparenz garantiert “[that] we can replace any expression in a program with its resulting value without changing the program’s semantics” (s. S.72, mittlere Spalte, Mitte) Referentielle Transparenz ist in rein funktionalen Sprachen durch die Abwesenheit von Seiteneffekten garantiert (abgesehen von Ausdrucken für Ein- und Ausgabe, die unvermeidbar seiteneffektbehaftet und in Sprachen wie Haskell deshalb speziell gekapselt sind).

# Vorlesungsfolien

## Haskell

* Große Community, große Bibliotheken, kommt mit standard module prelude
* Schnittstelle zu anderen Sprachen, zB zu C
* Rein funktionale Programmierung
* Statisch typisiert: Übersetzer, Interpretierer prüfen, ob die Typisierung von Haskell-Programmen wohlgetypt, konsistent ist.
* Typinferenz: Programmierer können Typdeklarationen angeben (aussagekräftigere Fehlermeldungen, Sicherheit), müssen aber nicht
* verzögerte Auswertung (lazy evaluation)
* Funktionen höherer Ordnung/Funktionale
* modernes polymorphes Typsystem, Polymorphie/Generizität
* Musterpassung (pattern matching)
* Modularisierung zum hochskalieren -> Datenabstraktion (abstrakte Datentypen)
* Formatierung des Programmtexts trägt Bedeutung (Tabs und Abstände machen einen Unterschied), dafür keine geschwungenen Klammern für Funktionen notwendig
* Haskell beruht auf angewandten typisierten λ-Kalkülen.
* Rekursion kann unmittelbar ausgedrückt werden (Y-Kombinator nicht erforderlich).

**Basics von Haskell**

**Relatoren**

>=, <=, ==, /=, usw

**Konstruktoren vs Operatoren**

Konstruktoren führen zu eindeutigen Darstellungen von Werten, Operatoren nicht.

* (:) ist (einziger) Konstruktor für Listen.
* (++) ist (einer von vielen) Operator(en) auf Listen.

[42,17,4] == (42:(17:(4:[]))) -- Eindeutige Darstellung von [42,17,4] mittels des Konstruktors

[42,17,4] == [42,17] ++ [] ++ [4] -- Viele Darstellungen mittels des Operators (++).

== [42] ++ [17,4] ++ []

== [42] ++ [] ++ [17,4] = ...

**Tupel**

(‘a‘,True) :: (Char,Bool)

* fassen eine vorbestimmte Zahl von Werten möglicherweise verschiedener Typen zusammen.
* heterogem
* Statt Tupeltyp sprechen wir auch von Kreuzprodukttyp.

fst, snd vordefiniert um Elemente anzusprechen (nur bei 2 Elementen). Man nennt sie „Selektoren für Tupeltypen“

**Listen**

* fassen eine nicht vorbestimmte Zahl von Werten gleichen Typs zusammen. Können unendlich sein
* homogen
* Nur typgleiche Listenwerte sind vergleichbar

Hinweis: Die Listenelementzugriffsfunktion (!!) hat Typ Int statt Integer

(!!) :: [a] -> Int -> a

**Vordefinierte Funktionen auf Listen**

Transformieren aller Listenelemente mittels einer Abbildungsvorschrift:

map :: (a -> b) -> [a] -> [b]

Herausfiltern aller Listenelemente mit einer bestimmten Eigenschaft:

filter :: (a -> Bool) -> [a] -> [a]

eigentlich äquivalent zu:

filter p ls = [ l | l <- ls, p l ]

Aggregieren aller Listenelemente mittels einer Verknüpfungsoperation:

foldr :: (a -> b -> b) -> b -> [a] -> b

foldl :: (a -> b -> a) -> a -> [b] -> a

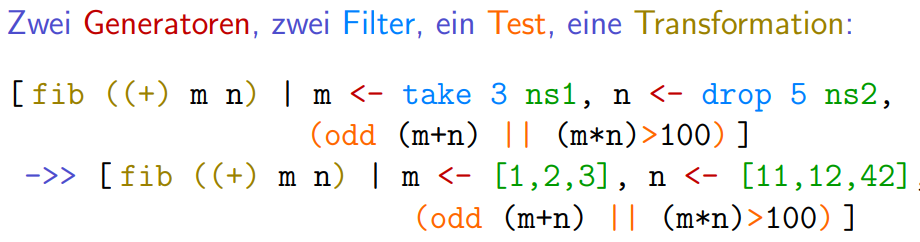
Die Faltungsfunktionale foldr und foldl unterscheiden sich hinsichtlich

* Anwendbarkeit (foldr, foldl mit Faltungsfunktionen (flip (:)), (:))
* Effizienz (foldr, foldl mit Faltungsfunktion (++))

Siehe Folien, Seite 836

**Listenkomprehension (list comprehension)**

* Generatoren
* Filter
* Tests
* Transformationen



**Funktionen**

Es werden bei den Funktionssignaturen und Funktionstermen Klammern ausgelassen:

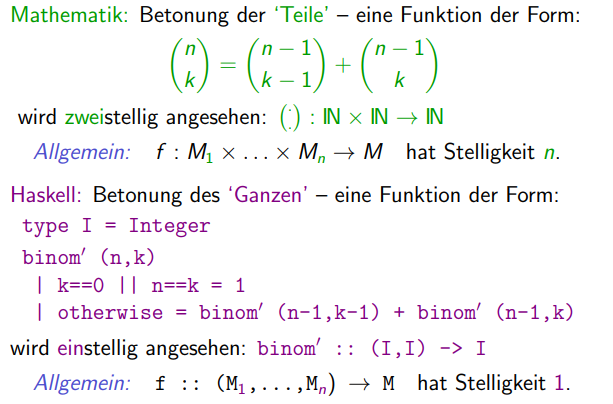
* Funktions-Signaturen sind *rechtsassoziativ* geklammert.
* Funktions-Terme sind *linksassoziativ* geklammert.
* Haskell-Funktionen sind einstellig: Es wird stets ein Argument zur Zeit konsumiert (kann aber auch Tupel sein)

Man sagt: Funktions-Stelligkeit ist 1.

* Argumente und Werte von Funktionen und Funktionstermen – können elementaren, zusammengesetzten (Tupel) oder funktionalen Typs sein.

**Funktions-Stelligkeit in Haskell vs. in der Mathematik**

Hinweis: hier für mathematischen Kreuzprodukt Tupel verwendet.



**Signaturen (Rechtsassoziativ)**

Syntaktische Funktionssignaturen geben den Typ einer Funktion an.

Rechtsassoziativität füt Funktionssignaturen

ersetze :: (String -> (Int -> (String -> (String -> String))))

Es wird mit jedem Schritt / jeder Argumentkonsumation eine Funktion gebildet der das nächste Argument konsumiert

**Funktionsterme (Linksassoziativ)**

Funktionsterme sind aus Funktionsaufrufen aufgebaute Ausdrücke.

Funktionssignatur

ersetze :: String -> Int -> String -> String -> String

Funktionsterme

(ersetze "Ein alter Text") :: (Int -> (String -> (String -> String))))

Dadurch, dass „ersetze“ nur ein Argument konsumiert, ist die Ausgabe eine Funktion

(ersetze "Ein alter Text" 1)

(ersetze "Ein alter Text" 1 "alter")

(ersetze "Ein alter Text" 1 "alter" "neuer")

Linksassoziativität für Funktionsterme:

ersetze "Ein alter Text" 1 "alter" "neuer"

steht eigentlich für:

((((ersetze "Ein alter Text") 1) "alter") "neuer")

**Fallunterscheidungen in Funktionen**

1. Bedingte Ausdrücke (if . then . else .)

Vorteile: Keine (oberflächliche syntaktische Ähnlichkeit mit Konstrukten aus imperativen Programmiersprachen)

Nachteile: Geschachtelt sind sie schwer zu lesen und verstehen.

1. Bewachte Ausdrücke (| waechterausdruck = …)

Vorteile: Kurz und klar, wenig syntaktischer Overhead.

Nachteile: mit let und in nicht gut kombinierbar

1. Musterbasierte Ausdrücke ((x:y:zs) = ausdruck)

Vorteile: Kurz und klar, wenig syntaktischer Overhead, praktisch bei strukturierten Argumenten, da die umständliche Verwendung von Selektorfunktionen dadurch vermieden werden kann, um auf die Strukturelemente zuzugreifen.

Nachteile: Schlechte Wartbarkeit, Änderungen aufwändig

1. case-Ausdrücke (case musterausdruck of wert -> ausdruck...)

Vorteile: Keine (auch mit musterbasierten Ausdrucken erreichbar)

Nachteile: Notationell aufwändiger als musterbasierte Ausdrücke

Unterschiede:

1,2 arbeiten mit Boolean aber 3,4 arbeiten mit der Struktur (vor allem bei Datenkonstruktoren).

## Berechenbarkeit

**Vereinfachte Vorstellung von Berechenbarkeit**

‘Etwas’ ist *intuitiv* berechenbar, wenn es eine ‘irgendwie machbare’ effektive mechanische Methode gibt, die für

1. gültige Argumentwerte: in endlich vielen Schritten den Funktionswert konstruiert
2. nicht gültige Argumentwerte: mit einem besonderen Fehlerwert oder nie abbricht

Jetzt zu definieren:

* ‘etwas’
* ‘irgendwie machbar’
* effektive mechanische Methoden M

Jede Methode M definiert ein formales Berechenbarkeitsmodell: „Berechenbar mit M bzw. M-berechenbar“.

Danach stellen wir die These auf:

M-These

‘Etwas’ ist M-berechenbar <=> ‘Etwas’ ist *intuitiv* berechenbar

(Denn es ist selbstverständlich, dass A <= B aber A => B lässt sich nicht beweisen.)

Das ist eine These und kein Theorem weil sie noch nicht belegt wurde. Sie ist aber falsifizierbar und bis dahin nehmen wir sie an.

Anstatt die These zu beweisen sagt man sie ist so lange gültig bis sie falsifiziert wurde.

Falsifizierbarkeit der M-These für jedes M:

Die M-These, ist widerlegt, wenn eine berechnungsstärkere, ausdruckskräftigere Methode M´ als M gefunden wird, wenn also etwas M´-berechenbar ist, aber nicht M-berechenbar.

**Berechnungsmodelle**

Ein Berechnungsmodell ist ein Kalkül. Berechnungsmodelle arbeiten mit Symbolen, Zahlen und Termen.

Sie sind formale Systeme mit eigenen Axiomen (Regeln) und dienen als mathematische Berechnungsgrundlage (formale Grundlage) für jede Programmiersprache.

Ein Kalkül muss in sich konsistent und vollständig sein sein (widerspruchsfrei)

Zusätzlich erforderlich:

* Turing-vollständig sein (alles berechnen können was mit einer Turingmaschine berechenbar ist)
* Praktische Umsetzbarkeit zu Programmiersprache

**Beispiele von Berechenbarkeitsmodellen**

* Allgemein / primitv rekursive Funktionen (Herbrand 1931, Gödel 1934, Kleene 1936)

Gehen von einer vorgegebenen Menge an einfachen Funktionen aus. Komposition von anderen Funktionen oder sich selbst (Rekursion) bildet neue Funktionen.

Nicht alleine Turing-vollständig -> Turing-Vollständigkeit erreicht durch μ-rekursive Funktionen und λ-Kalkül

Sie galten bis 1928 als Turing-vollständig, aber sie sind schwächer als alle anderen Modelle.

Falsifizierung der Primitiv-rekursiv-These durch Wilhelm Ackermann (1928) durch Angabe der später nach ihm benannten – Ackermann-Funktion die berechenbar ist, aber nicht primitiv-rekursiv-berechenbar.

* µ-rekursive Funktionen (Kleene 1936)
* λ-definierbare Funktionen des λ-Kalküls (Church 1936)
* Turing-Maschinen (Turing 1936)
* Endliche kombinatorische Prozesse (Post 1936)
* Markov-Algorithmen (Markov 1951)
* Registermaschinen (Random Access Machines (RAMs)) (Shepherdson, Sturgis 1963)

**Theorem - Gleichmächtigkeit**

Alle der vorher genannten formalen Berechnungsmodelle und ihre zugehörigen (effektiven mechanischen Berechnungs-) Methoden sind gleich mächtig:

* Was in einem dieser Modelle berechenbar ist, ist in jedem der anderen Modelle berechenbar und umgekehrt!

Daraus folgt : Universalität des λ-Kalküls

Alles, was in einem der vorher genannten Modelle berechenbar ist, ist im λ-Kalkül berechenbar und umgekehrt!.

Dieses Theorem schließt nicht aus, dass vielleicht noch heute eine mächtigere (Berechnungs-) Methode gefunden wird, die ‘etwas’ und *intuitiv berechenbar* umfassender, vollständiger ist. Dadurch möglich, dass nur ein Problem dass offensichtlich berechenbar ist, aber nicht im λ-Kalkül.

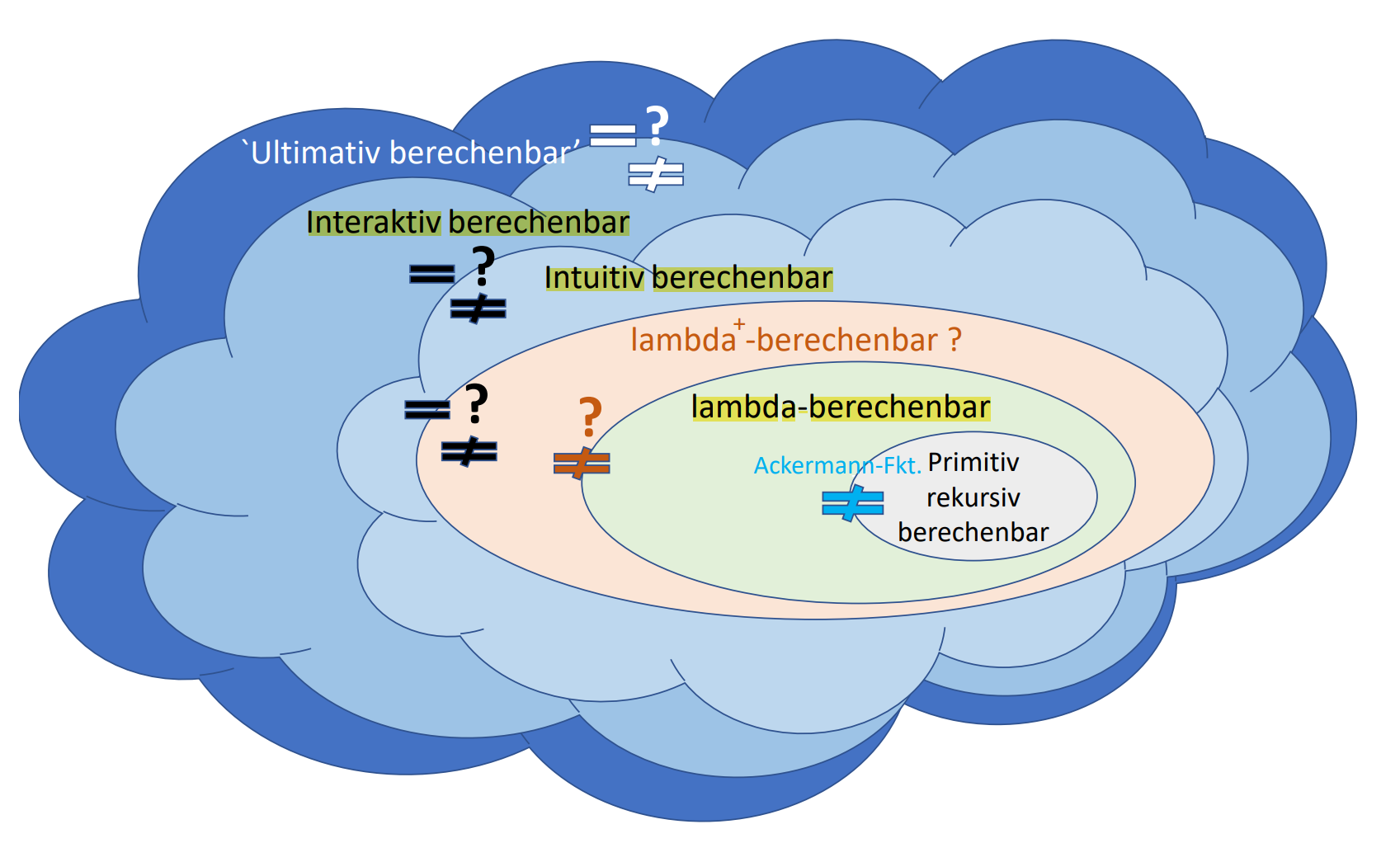
**Churchsche These (λ-Kalkülthese)**

‘Etwas’ ist intuitiv berechenbar <=> es ist im λ-Kalkül berechenbar.

**Umfasst ‘etwas’ vollständig alle Probleme?**

Wir haben eine funktionsorientierte Vorstellung von ‘etwas’ die nur auf Eingaben und Ausgaben basiert ist. Dadurch ist die Zustandslosigkeit von funktionaler Programmierung kein Problem. -> Dadurch, dass ‘etwas’ Interaktion ausschließt, beschränkt es implizit die Problemtypen.

Ändert Hinzunahme von Interaktion das Verständnis von Berechenbarkeit wie die Findung der Ackermann-Funktion?



## λ-Kalkül (Berechnungsmodell)

* Alonzo Church
* formales (ursprünglich naturwissenschaftliches) Berechnungsmodell, wurde zur Grundlage aller funktionalen Programmiersprachen -> Bindeglied funktionaler Hochsprachen und maschinennaher Implementierungen.
* formalisiert einen Berechnungsbegriff über Paaren, Listen, Bäumen, auch potentiell unendlichen, über Funktionen

höherer Ordnung, etc.

Besonders:

* Einfachheit: wenige syntaktische Konstrukte, einfache Semantik.
* Ausdruckskraft: Turing-mächtig, alle ‘intuitiv berechenbaren’ Funktionen sind im λ-Kalkül berechenbar.

**Anwendungsgebiete**

Ursprünglich:

* Berechenbarkeitstheorie: Berechenbarkeitsbegriff, Grenzen der Berechenbarkeit.

Später hinzugekommen:

* Entwurf von Programmiersprachen und Programmiersprachkonzepten: Funktionale Programmiersprachen, Typsysteme, Polymorphie,...
* Semantik von Programmiersprachen: Denotationelle Semantik, Bereichstheorie (engl. domain theory),...

**Reiner λ-Kalkül**

reduziert auf das ‘absolut Notwendige’, besonders wichtig für Berechenbarkeitstheorie.

Syntax des reinen λ-Kalküls nennt man kurz λ-Ausdrücke.

Semantik / Bedeutung / Wert von λ-Ausdrücken:

* eindeutig bestimmte Normalform, wenn sie existiert
* undefiniert, wenn die Normalform nicht existiert

**angewandte λ-Kalküle**

Syntaktisch erweiterte Varianten des reinen λ-Kalküls, praxis- und programmiersprachennäher.

Extrem angereicherte angewandte λ-Kalküle nennt man Funktionale Programmiersprachen.

### Reiner λ-Kalkül

**λ-Ausdrücke**

Die Menge E (von engl. Expression) der Ausdrücke des reinen λ-Kalküls über einer Menge N von Namen ist induktiv wie folgt definiert:

* ***λ-Namen:*** Variablen (für Werte oder Funktionen)

Man benutzt auch statt dem Namen aus dem Namensraum N auch die Bezeichnung Variable aus dem Variablenraum V.

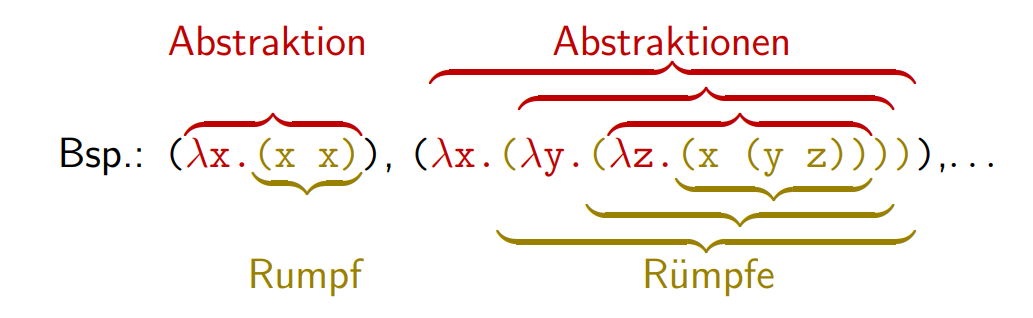
Jeder Name aus N ist in E.

Bsp.: a, b, c, . . . , x, y, z, . . .

* ***λ-Abstraktionen:*** Anonyme Funktionen, Lambda Ausdrücke

Ist x aus N, e aus E, so ist (λx. e) in E.

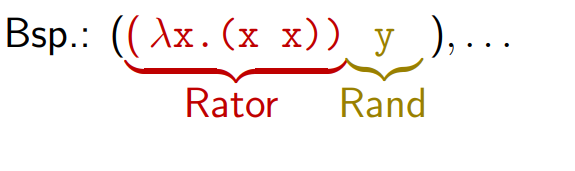
Sprechweise: (Funktions-) Abstraktion mit Parameter x und Rumpf e.



* ***λ-Applikationen:*** Funktionsanwendungen, Funktionsterme mit Eingabe

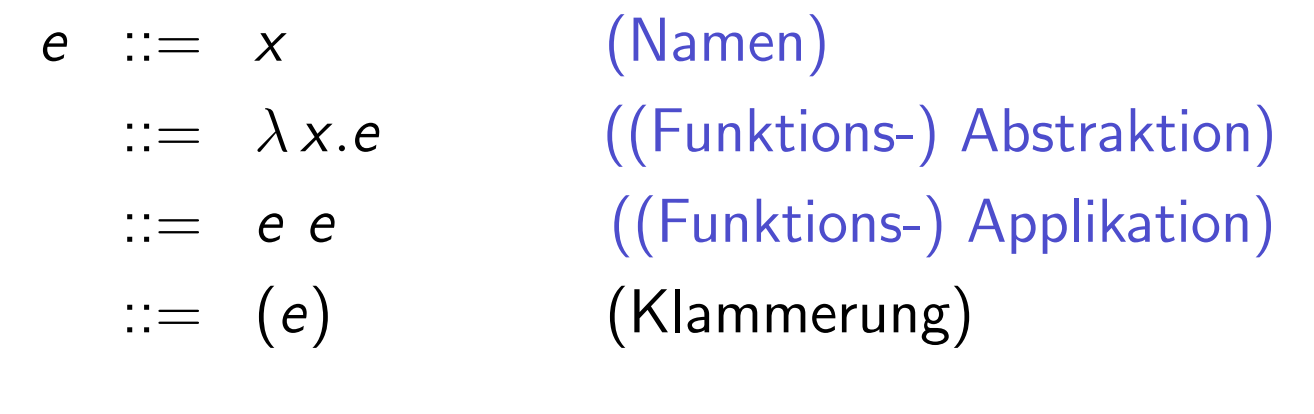
Sind f und e aus E, so ist auch (f e) in E.

Sprechweise: Applikation oder Anwendung von f auf e. f heißt auch Rator, e auch Rand.



Der Rand ist das Argument, das von der Lambda Funktion konsumiert wird.

**Backus-Naur-Form (BNF) Notation:**



**Rechts und Links-Assoziativität**

Überflüssige Klammern können weggelassen werden.

* *Rechtsassoziativität für λ-Sequenzen in Abstraktionen.*

λx.λy.λz.(x (y z)) gleich mit (λx.(λy.(λz.(x (y z)))))

λx. e gleich mit (λx. e)

* *Linksassoziativität für Applikationssequenzen.*

e1 e2 e3 ... en gleich mit (((e1 e2) e3)... en)

e1 e2 gleich mit (e1 e2)

**Freie und gebundene Variablen**

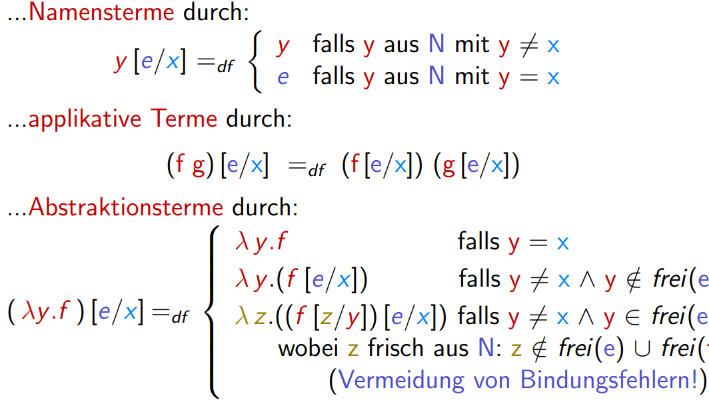
Eine Variable ist gebunden wenn es unmittelbar nach einem λ folgt. Ansonsten ist es ungebunden.

((λx. (x y)) x)

x kommt frei und gebunden vor (einmal als Eingangsparameter, einmal einfach am Rand)

y kommt nur frei vor.

#### Syntaktische Substitution

Ist eine dreistellige Abbildung

·[·/·] : E → E → V → E

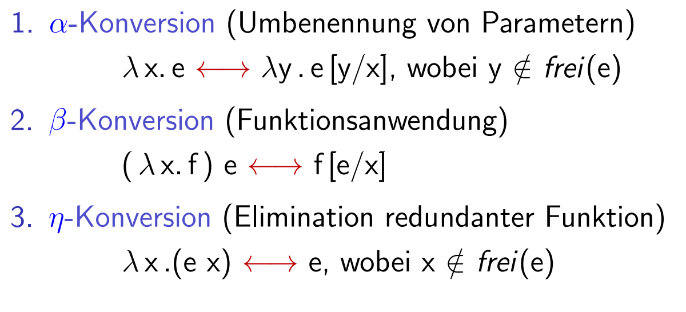
Vereinfacht: Angewendet auf zwei Ausdrücke e´ und e und eine Variable x bezeichnet e´ [e/x] denjenigen Ausdruck, der aus e´ entsteht, indem jedes freie Vorkommen von x in e´ durch e substituiert / ersetzt wird.

e´ [e/x] (Alle x aus e´ werden zu e)

Beispiele:

* ((x y) (y z)) [(a b)/y] = ((x (a b)) ((a b) z))
* λx. (x y) [(a b)/y] = λx. (x (a b))
* λx. (x y) [(a b)/x] = λx. (x y)

#### Reduktions und Konversionsregeln

Ziel: maximale Ausdrucksvereinfachung mit syntaktischer Konversion

α-Konversion

* Reine Umbenennung von gebundenen Parametern in λ-Abstraktionen (zur Vermeidung von Bindungsfehlern bei β-Konversion )

β-Konversion

* Funktionsanwendung - zur Anwendung einer λ-Abstraktion auf ein Argument -> Argument wird von außen nach innen geholt

η-Konversion

* zur Elimination unnötiger λ-Abstraktionen -> Funktionen entfernen die Redundant sind

**α-Konversion um Bindungsfehler zu vermeiden**

Ein Bindungsfehler entsteht wenn ein Ausdruck mit einer freien Variable in den Gültigkeitsbereich einer gebundenen Variable gleichen Namens eingesetzt wird:

1. (λx. x y) [x/y] −−naiv−→ (λ x. x x)

2. (λx. (x y) [(x b)/y] −−naiv−→ λx. (x (x b))

Dadurch hat aber Eingangsparameter direkten Einfluss nach Ersatz, da x gebunden in Ausdruck. Korrekt mit Umbenennung angewendet gibt es keinen Bindungsfehler:

1. (λ x. x y) [x/y] =

(λ z. x y) [z/x]) [x/y]

(λ z. z y)[x/y]

λ z. z x

2. λx. (x y) [(x b)/y] =

λz. ((x y)[z/x]) [(x b)/y] Umbenennung von x in z, damit (x b) frei bleiben kann

λz. (z y) [(x b)/y]

λz. (z ( x b)) Dadurch bleibt x in (x b) frei

**Unterscheidung zwischen Konversion und Reduktion**

Sprechweisen im Zusammenhang mit Konversionsregeln:

* *Von links nach rechts angewendet: Reduktion* -> β-Reduktion und η-Reduktion
* *Von rechts nach links angewendet: Abstraktion*

**Maximale Reduktionsfolge / Normalform**

Eine Reduktionsfolge für einen λ-Ausdruck

* ist eine endliche oder nicht endliche Folge von β-, η-Reduktionen und α-Konversionen.
* *Reduktionsfolge* heißt *maximal*, wenn höchstens noch α-Konversionen anwendbar sind.
* Ein *λ-Ausdruck* ist in *Normalform*, wenn er durch β-, η-Reduktionen nicht weiter reduzierbar ist.

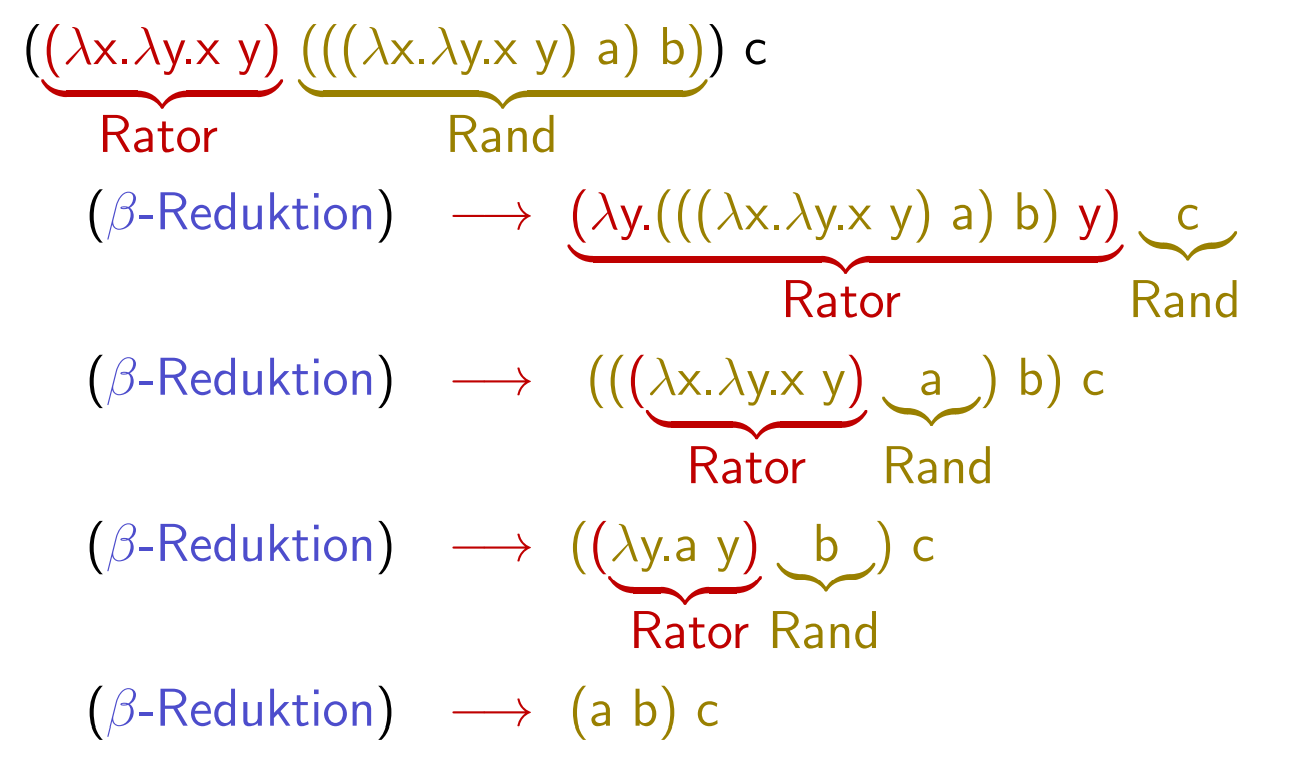
Wozu die Normalform dient:

λ-Ausdrucke in Normalform dienen dazu, die Semantik, die Bedeutung von λ-Ausdrucken zu definieren, sie auszuwerten.Die Bedeutung eines λ-Ausdrucks ist seine bis auf α-Konversionen eindeutig bestimmte Normalform, wenn sie existiert.

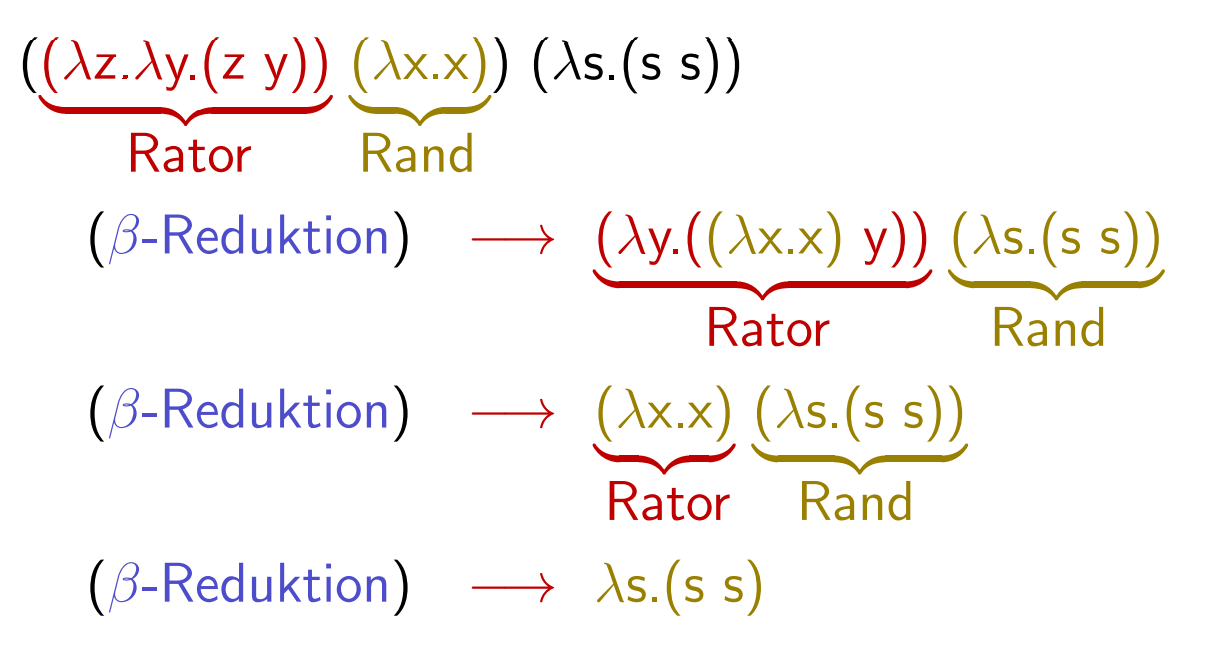
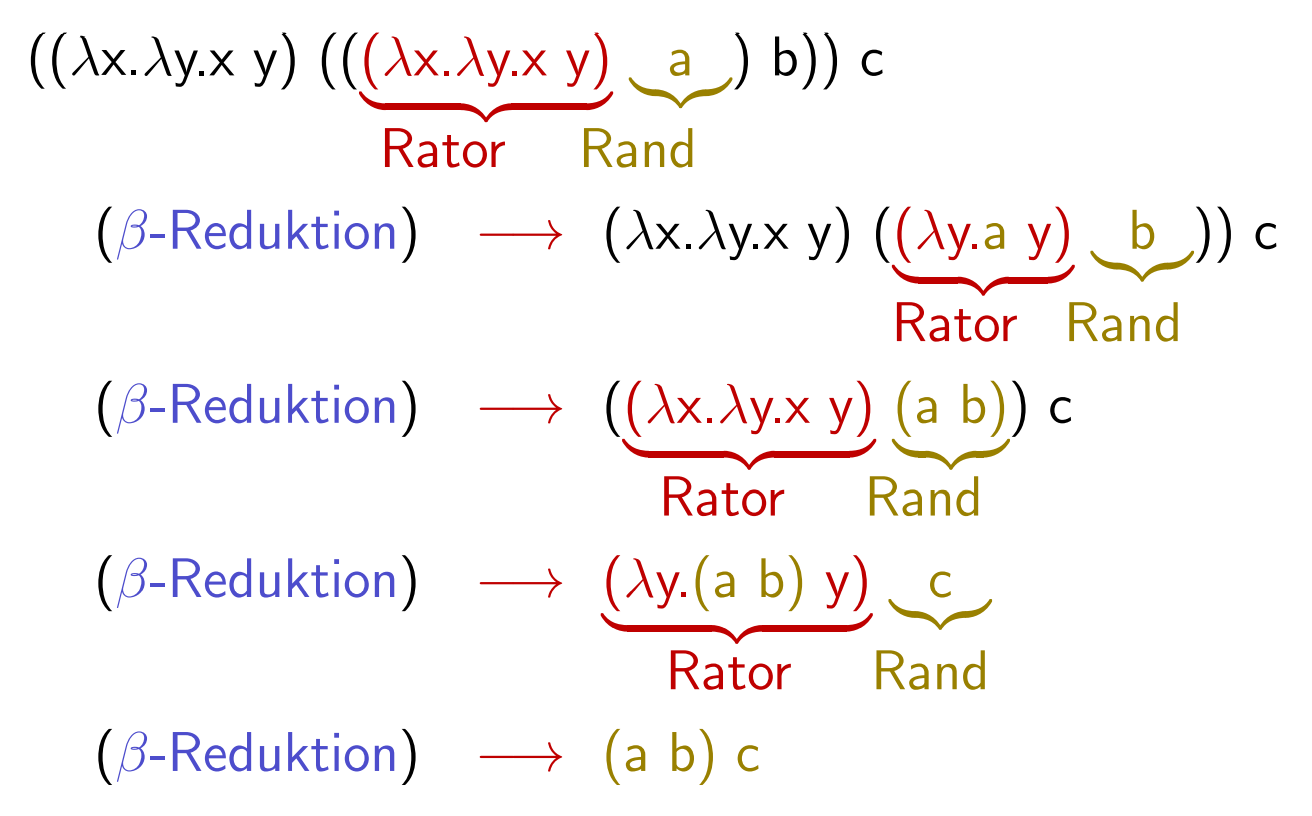
Es gibt 2 Reduktionsstrategien / Reduktionsreihenfolgen um die Normalform zu erreichen:

* Links-*Normale* Reduktionssordnung (linkest-äußerst)
* Links-*Applikative* Reduktionssordnung (linkest-innerst)

Beispiel: Normale Ordnung



Beispiel: Applikative Ordnung

**Terminierung der Reduktionsfolgen**

* Nicht jede Reduktionsfolge teminiert -> Endlosschleife von β-Reduktionen möglich (zB durch Y-Combinator)
* Nicht jeder λ-Ausdruck besitzt eine Normalform
* Reduktionsordnung (applikative / normale reduktion) kann entscheiden ob die existierende Normalform erreicht wird

#### Die Church/Rosser-Theoreme

Seien e1, e2 zwei λ-Ausdrücke.

**1. Theorem - Konfluenz-, Diamant-, Rauten-Eigenschaft**

Wenn die Normalform eines λ-Ausdrucks existiert, ist sie (bis auf gebundenes Umbenennen von Namen) eindeutig bestimmt.

Wenn e1, e2 ineinander konvertierbar sind dann gibt es einen gemeinsamen Ausdruck e, zu dem beide reduziert werden können.

e1 → e2 bedeutet e1 → e → e2

e1 ← e2 bedeutet e1 ← e ← e2

dadurch:

e1 ←→ e2 bedeutet e1 → e und e2 → e

**2. Theorem - Standardisierung**

Wenn irgendeine Reduktionsfolge zur Normalform eines λ-Ausdruck führt, dann auch immer die normale Reduktionsordnung.

Anders ausgedrückt: Die Auswertung gemäß normaler Reduktionsordnung terminiert am häufigsten.

**3. Daraus folgt: Determiniertheit der Konversion, Turing-Mächtigkeit**

Wenn ein Ausdruck auf Normalform reduziert werden kann, dann führt jede terminierende Reduktionsfolge zu dieser Normalform. Alle Reduktionen führen zum selben determinierten Ergebnis.

Damit ist λ-Ausdrucken in Form ihrer Normalform in eindeutiger Weise eine Bedeutung gegeben.

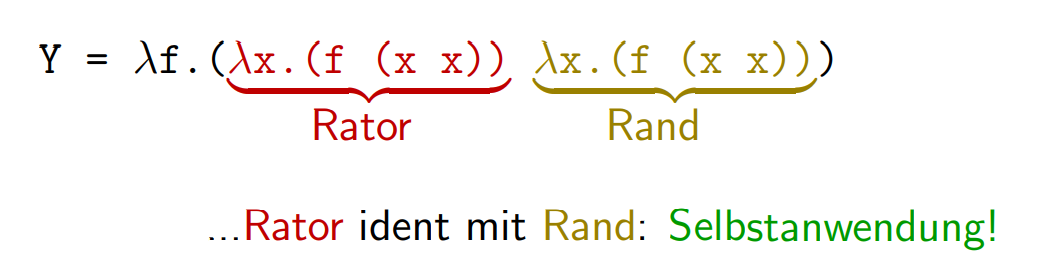
#### Rekursion und der Y-Kombinator

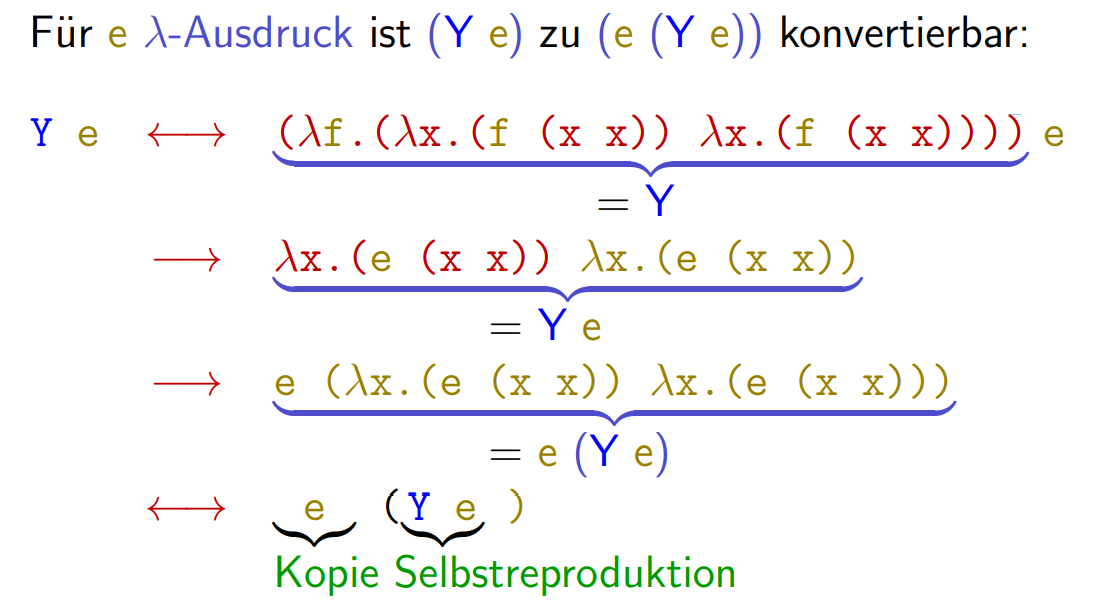
λ-Abstraktionen sind anonym und können deshalb nicht (rekursiv) aufgerufen werden. Rekursion ist im reinen λ-Kalkül nicht möglich.

**Kombinatoren: der Y-Kombinator**

Lambda-Terme ohne freien Variablen heißen Kombinatoren.

Der Y-Kombinator ist ein Kombinator mit Selbstverwendung und der Fähigkeit sich selbst zu reproduzieren und die Argumente zu kopieren. Das ermöglicht Rekursion.





In Haskell:

fixpoint f = (\f -> (\x -> f (x x)) (\x -> f (x x)))

fixpoint f = f(fixpoint f)

y = \f -> (\x -> f (x x)) (\x -> f (x x))

-- man ruft den Y-Combinator mit der Funktion f die rekursiv aufgerufen werden soll auf

>> y = \f -> (\x -> f (x x)) (\x -> f (x x)) f

>> y = (\x -> f (x x)) (\x -> f (x x))

>> y = f ((\x -> f (x x) (\x -> f (x x))

>> y = f (f ((\x -> f (x x) (\x -> f (x x))))

>> y = f (f (f ((\x -> f (x x) (\x -> f (x x)))))

>> y = f (f (f (f ((\x -> f (x x) (\x -> f (x x))))))

...

>> y = f(f(f(f(...))))

-- Vereinfacht entspricht der Y-Combinator dem Aufruf:

rec f = f(rec f)

>> f(rec f)

>> f(f(rec f))

>> f(f(f(rec f))

>> f(f(f(f(rec f))

...

>> f(f(f(f(...))))

#### Factorial mit Y Combinator

Der nicht rekursive Teil im Y Combinator wird mit f übergeben. Daher können wir für eine bestimmte Funktion die Berechnungs- und Abbruchbedingungen über diese Funktion bestimmen. Hier bestimmen wir `f` für die Factorial Funktion.

fac :: ((a -> b) -> a) -> b

fac = (\g -> (\n -> if n == 1 then 1 else n \* g(n-1)))

-- 2 Argumente benötigt: eine Funktion und eine Zahl

-- Aufruf ohne Y-Combinator um 2 Fakultät zu berechnen: fac fac 2

Dabei ist `g` die Funktion der Rekursion durch `x` des Y Combinator.

Ich versuche die Auswertung von `y fac 2` zu skizzieren:

y f = (\x -> f (x x)) (\x -> f (x x))

y fac = (\x -> fac (x x)) (\x -> fac (x x))

y fac 2 = (\x -> fac (x x)) (\x -> fac (x x)) 2

(\f -> (\x -> f (x x)) (\x -> f (x x))) fac 2

(\x -> fac (x x)) (\x -> fac (x x)) 2

fac (\x -> fac (x x)) (\x -> fac (x x)) 2

(\g -> \n -> if n == 1 then 1 else n \* g(n-1)) ((\x -> fac (x x)) (\x -> fac (x x))) 2

(\n -> if n == 1 then 1 else n \* ((\x -> fac (x x)) (\x -> fac (x x))) (n-1)) 2

if 2 == 1 then 1 else 2 \* ((\x -> fac (x x)) (\x -> fac (x x))) (2-1)

2 \* ((\x -> fac (x x)) (\x -> fac (x x))) 1

2 \* fac ((\x -> fac (x x)) (\x -> fac (x x))) 1

2 \* (\g -> \n -> if n == 1 then 1 else n \* g(n-1)) ((\x -> fac (x x)) (\x -> fac (x x))) 1

2 \* (\n -> if n == 1 then 1 else n \* ((\x -> fac (x x)) (\x -> fac (x x))) (n-1)) 1

2 \* if 1 == 1 then 1 else 2 \* ((\x -> fac (x x)) (\x -> fac (x x))) 1

2 \* 1

2

Dieser code würde nicht kompilieren, weil der Aufruf `x x` unendlich wäre. Mit einer extension würde das gehen, aber standardmäßig nicht.

### Angewandte λ-Kalküle

sind syntaktisch angereicherte Varianten des reinen λ-Kalküls. In Ausdrücken angewandter λ-Kalküle:

* Konstanten, Funktionsnamen, übliche Operatoren ähnlich wie Namen

42, 3.14, true, false, +, ∗, −, fac, binom,…

* neue Ausdrücke als Abkürzungen eingeführt und verwendet

cond e e1 e2 , if e then e1 else e2,…

* Typen auftreten, Ausdrücke getypt sein

42 : IN, 3.14 : IR, true : IBool,…

* Applikative Terme

2+3, fac 3, fib (2+3), binom x y, ((binom x) y),…

* Abstraktionsterme

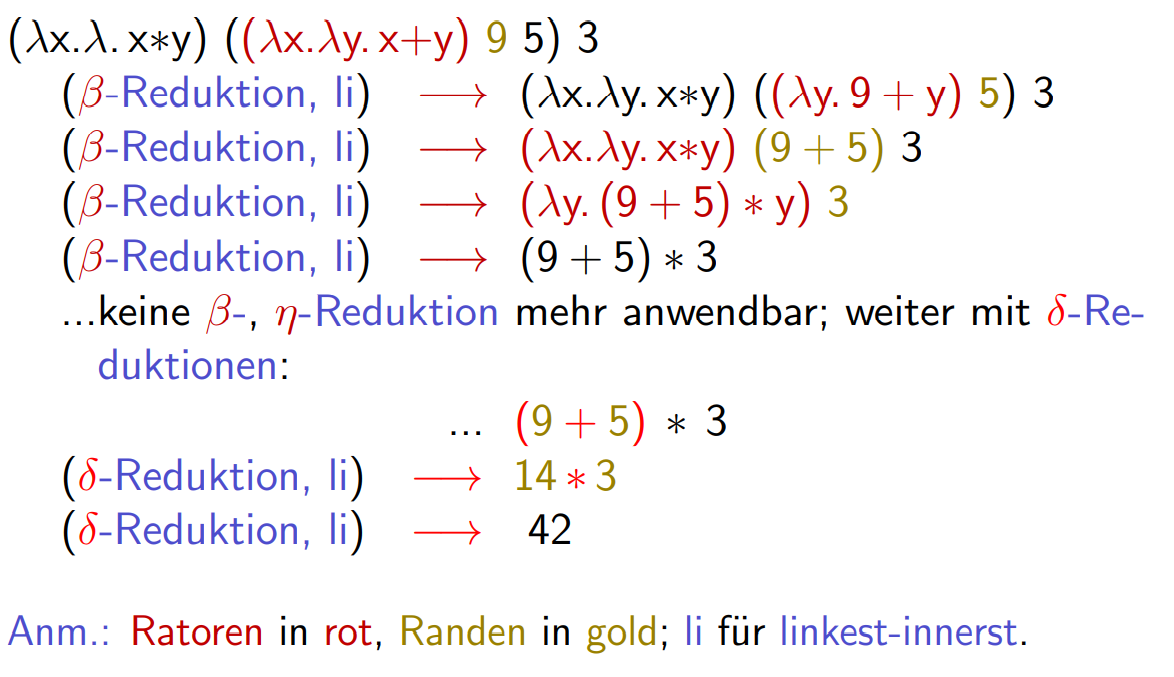
λx. (x + x), λx.λy.λz. (x ∗ (y − z)),

(λx. if odd x then x ∗ 2 else x div 2),…

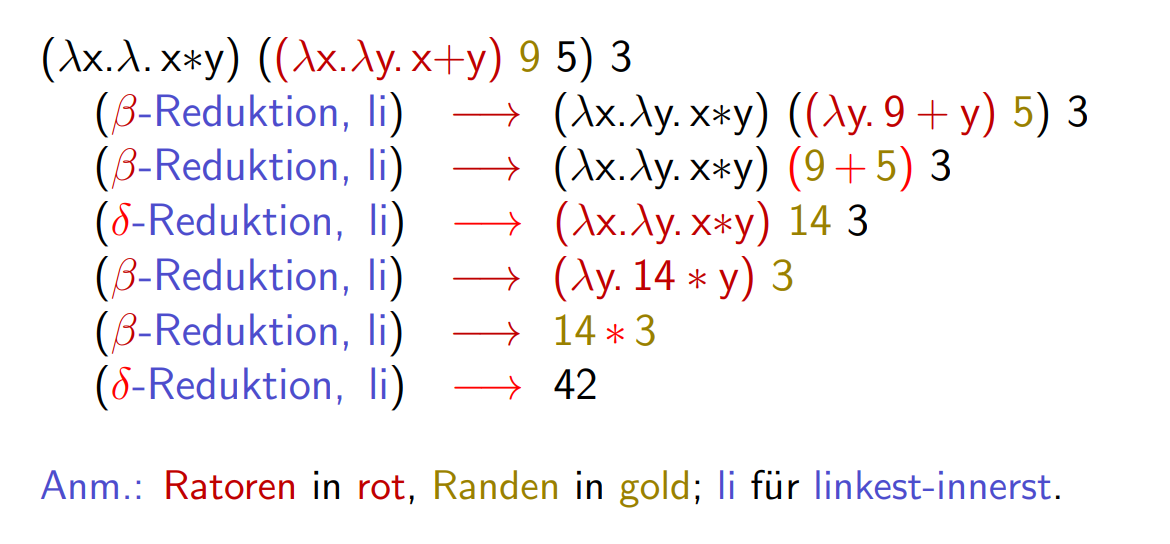
**δ-Reduktionsfolge**

Dadurch zusätzliche Reduktionsregeln nötig: δ-Reduktionen für die Auswertung, Reduktion arithmetischer Ausdrücke, bedingter Ausdrücke, Operationen auf Listen, etc.

δ-Reduktionsfolge: Unecht applikative Ordnung (unecht, da ohne β-, η- und δ-Reduktionen)



δ-Reduktionsfolge: applikative Ordnung



Typisierte λ-Kalküle

ordnen jedem wohlgeformten Ausdruck einen Typ zu, z.B.:

3 : Int

(∗) : Int → Int → Int

(λx.2 ∗ x) : Int → Int

(λx.2 ∗ x) 3 : Int

Ausdrücke mit Selbstanwendung wie der Y-Kombinator erzeugen 2 Schwierigkeiten:

* *Typisierung*: haben keinen endlichen Typ, ihr Typ ist nicht durch einen gewöhnlichen endlichen Typausdruck beschreibbar.

Lösung: Übergang zu mächtigeren Typsprachen (Bereichstheorie, reflexive Bereiche (engl. domain theory, reflexive domains)).

* *Rekursion*: können nicht zur Modellierung von Rekursion verwendet werden.

Lösung: Explizite Hinzunahme der Reduktionsregel Y e → e (Y e) zum Kalkül.

Lösungen lassen sich rechtfertigen aus theoretischer Sicht.

## Auswertung von Ausdrücken

Ein Ausdruck ist in Programmiersprachen ein Konstrukt, dass in Bezug auf einen Kontext ausgewertet werden kann, also einen Wert liefert.

Arithmetische Ausdrücke: 2\*3; 2(x − a); x2 = 2x; y= sin(x);

Aussagenlogische Ausdrücke: a und b; wenn b dann a; a oder nicht b;

Prädikatenlogische Ausdrücke: wenn verheiratet(A, B) dann verheiratet(B, A);

Ausdrücke in Programmiersprachen

* Literale (Konstanten): 2, 3.14, Strings und Zeichen …
* Variablen: x, y, z, …
* Funktionen: sin(phi), random(), aktMonatsNummer(), …
* Operationen: 2\*3, 2(x - a), x^3, cos(2\*x + 3.14), …

**Expansions und Simplifikations-Schritte beim Auswerten**

* *Expansions-Schritte:* Funktionsterme, Funktionsaufrufe
* *Simplifikations-Schritte:* Funktionstermfreie Ausdrücke Die einzelnen Vereinfachungs-, Rechenschritte

Beispiel:

Auswertung mit Funktionstermen - Durch den Aufruf einer Funktion ist eine Expansion notwendig.

simple x y z :: Int -> Int -> Int -> Int

simple x y z = (x + z) \* (y + z)

Der Funktionsterm simple 2 3 4 hat als Semantik / Bedeutung den Wert 42:

1. Weg:

simple 2 3 4

(Expandieren) ->> (2 + 4) \* (3 + 4)

(Simplifizieren) ->> 6 \* (3 + 4)

(S) ->> 6 \* 7

(S) ->> 42

2. Weg:

simple 2 3 4

(E) ->> (2 + 4) \* (3 + 4)

(S) ->> (2 + 4) \* 7

(S) ->> 6 \* 7

(S) ->> 42

### Auswertungs-Reihenfolgen / -Ordnungen

Manchmal hat man bei der Berechnung von Funktionstermen verschiedene Möglichkeiten der Auswertung.

Ausdrucks-Auswertung =/= Ausdrucks-Vereinfachung mit syntaktischer Konversion.

Die Auswertungsreihenfolge hat keinen Einfluss auf den Wert des Ausdrucks.

* **Links Applikative Auswertung**: Linkest-innerste Stelle, Ausgewertet übergeben, Argumentauswertung vor Expansion des Funktionsaufrufs
* **Links Normale Auswertung**: Linkest-äußerste Stelle, Unausgewertet übergeben, Argumentauswertung nach Expansion des Funktionsaufrufs

nur sinnvoll wenn:

* *mit Ausdrucksteilung (engl. expression sharing)* als Implementierungstrick.

Dadurch späte, faule Auswertung (engl. lazy evaluation).

Beispiel:

fac :: Integer -> Integer

fac n = if n == 0 then 1 else (n \* fac (n - 1))

A): Applikativ – Argumentauswertung vor Expansion

fac 2

(E) ->> if 2 == 0 then 1 else (2 \* fac (2 - 1))

(S) ->> if False then 1 else (2 \* fac (2 - 1))

**Ab hier:** **(S) ->> (2 \* fac (2 - 1))**

(S) ->> 2 \* fac 1

(E) ->> 2 \* (if 1 == 0 then 1 else (1 \* fac (1 - 1)))

(S) ->> 2 \* (if False then 1 else (1 \* fac (1 - 1)))

(S) ->> 2 \* ((1 \* fac (1 - 1)))

(S) ->> 2 \* (1 \* fac 0)

(E) ->> 2 \* (1 \* (if 0 == 0 then 1 else (0 \* fac (0 - 1))))

(S) ->> 2 \* (1 \* (if True then 1 else (0 \* fac (0 - 1))))

(S) ->> 2 \* (1 \* 1)

(S) ->> 2 \* 1

(S) ->> 2

B): Normal – Argumentauswertung nach Expansion

fac 2

(E) ->> if 2 == 0 then 1 else (2 \* fac (2 - 1))

(S) ->> if False then 1 else (2 \* fac (2 - 1))

**Ab hier: (S) ->> (2 \* fac (2 - 1))**

(E) ->> 2 \* (if (2-1) == 0 then 1 else ((2-1) \* fac ((2-1)-1)))

(2S)->> 2 \* (if False then 1 else ((2-1) \* fac ((2-1)-1)))

(S) ->> 2 \* (((2-1) \* fac ((2-1)-1)))

(S) ->> 2 \* (1 \* fac ((2-1)-1))

(E) ->> 2 \* (1 \* (if ((2-1)-1) == 0 then 1 else ((2-1)-1) \* fac (((2-1)-1)-1)))

(E) ->> 2 \* (1 \* (if True then 1 else ((2-1)-1) \* fac (((2-1)-1)-1)))

(3S)->> 2 \* (1 \* 1)

(2S)->> 2

Beispiel:

2^3+fac(fib(squ(2+2)))+3\*5+fib(fac(7\*(5+3)))+fib((5+7)\*2)

A): Linksapplikative Auswertung

2^3+fac(fib(squ(2+2)))+3\*5+fib(fac(7\*(5+3)))+fib((5+7)\*2)

->> 8+fac(fib(squ(2+2)))+3\*5+fib(fac(7\*(5+3)))+fib((5+7)\*2)

->> 8+fac(fib(squ 4))+3\*5+fib(fac(7\*(5+3)))+fib((5+7)\*2)

->> 8+fac(fib((4\*4)))+3\*5+fib(fac(7\*(5+3)))+fib((5+7)\*2)

->> 8+fac(fib 16)+3\*5+fib(fac(7\*(5+3)))+fib((5+7)\*2)

->> 8+fac(fib (16-2)+fib(16-1)) + 3\*5 +...

->> 8+fac(fib 14+fib(16-1)) + 3\*5 +...

->> ...

B): Linksnormale Auswertung

2^3+fac(fib(squ(2+2)))+3\*5+fib(fac(7\*(5+3)))+fib((5+7)\*2)

->> 8+fac(fib(squ(2+2)))+3\*5+fib(fac(7\*(5+3)))+fib((5+7)\*2)

->> 8+(if fib(squ(2+2))==0 then 1 else n\*fac(fib(squ(2+2))1))+3\*5+fib(fac(7\*(5+3)))+ fib((5+7)\*2)

->> 8+( if (if squ(2+2)==0 then 0 else if squ(2+2)==1 then 1 else

fib(squ(2+2)-2)+fib(squ(2+2)-1))==0

then 1 else n\*fac(fib(squ(2+2))-1)) + 3\*5 +...

->> 8+( if (if (2+2)\*(2+2)==0 then 0 else if squ(2+2)==1 then 1

else fib(squ(2+2)-2)+fib(squ(2+2)-1))==0

then 1 else n\*fac(fib(squ(2+2))-1)) + 3\*5 +...

->> 8+( if (if 4\*(2+2)==0 then 0 else if squ(2+2)==1 then 1

else fib(squ(2+2)-2)+fib(squ(2+2)-1))==0 then 1 else n\*fac(fib(squ(2+2))-1))+ 3\*5 +...

->> ...

### Die Church/Rosser-Theoreme

**Theorem - Wertdeterminiertheit**

Jede terminierende Auswertungsfolge für einen Ausdruck endet mit demselben Ergebnis.

**Theorem – Terminierungshäufigkeit / Abweichendes Terminierungsverhalten**

Terminiert irgendeine Auswertungsfolge für einen Ausdruck, so terminiert auch seine (links-) normale Auswertung.

Die (links-) normale Auswertung eines Ausdrucks kann terminieren, während seine (links-) applikative nicht terminiert.

**Folgerungen:**

(Links-) applikative und (links-) normale Auswertungsordnung können sich bei der Auswertung unterscheiden in:

* Terminierungshäufigkeit

Normal Terminiert immer wenn terminiert werden kann im Gegensatz zur Applikativen Auswertung

* Terminierungsgeschwindigkeit / Performance

können bis zur Terminierung (mit gleichem Endresultat) unterschiedlich viele Expansions- und Simplifikationsschritte benötigen

nicht aber im:

* Ergebnis

Wenn beide terminieren, so terminieren sie mit demselben Resultat

### Lazy Evaluation: effiziente Implementierung der links-normalen Auswertung

Mehrfachauswertungen: Funktionsterm-Argumente werden (im worst case) so oft ausgewertet wie sie im Ausdruck vorkommen. Dadurch linksapplikative Auswertung effizienter.

**Lösung durch effizientere Implementierung:**

Ausdrucksdarstellung in Form von Graphen, wodurch gemeinsame Teil-Ausdrücke geteilt werden können *(engl. expression sharing)*:

* Ausdrucksauswertungen auf Graphen darstellen
* Wird ein Ausdruck ausgewertet, steht sein Wert an allen Verwendungsstellen zur Verfügung
* garantiert, dass Argumente höchstens einmal ausgewertet werden (möglicherweise auch gar nicht) - Erreicht annähernd gleiche Effizienz wie applikative Auswertung.
* Terminiert immer wenn terminiert werden kann

#### Übersetzeroptimierung durch Striktheitsanalysen

Striktheit: Definiertheit von Funktion bestimmt durch Argumente

**Striktheit von Funktionen**

Eine Funktion f heißt *strikt in ihrem n-ten* *Parameter* (oder Argument), wenn gilt: Ist der Wert des Arguments des n-ten Parameters nicht definiert, so ist auch der Wert von f nicht definiert (dadurch f nicht auswertbar unabhängig von den Werten möglicher weiterer Argumente).

Beispiel: Striktheit bei einstelligen Funktionen

Die Fakultäts- und Fibonacci-Funktion sind strikt in ihrem ersten (und einzigen) Parameter.

Undefiniertheit des Argumentwerts impliziert Undefiniertheit der Funktion.

fac (1 ‘div‘ 0) ->> undef

Beispiel: Striktheit bei mehrstelligen Funktionen

Mehrstellige Funktionen können strikt in einigen Parametern, nicht strikt in anderen sein:

if . then . else . ist strikt im 1-ten Argument (Bedingung), nicht strikt im 2-ten und 3-ten Argument

**Theorem – Striktheit und Terminierung**

Für strikte Funktionen stimmen die Terminierungsverhalten und Resultat von früher und später Auswertungsordnung für die strikten Argumente überein.

-> Durch den Übergang von normaler auf applikativer Auswertung für strikte Argumente einer Funktion gehen keine Ergebnisse verloren.

**Anwendung: Optimierung der Übersetzung von funktionalen Sprachen mit später Auswertung**

Durch frühe Auswertung für strikte Funktionsargumente die Übersetzung optimieren da bei applikativer Auswertung Zusatzaufwand für die Verwaltung geteilter Datenstrukturen fehlt.

**Striktheitsanalyse**

Übersetzer spät auswertender Sprachen führen dazu eine sog. Striktheitsanalyse durch:

* dort, wo es sicher ist, dass ein Ausdruck zum Ergebnis beiträgt und sein Wert deshalb in jeder Auswertungsordnung benötigt wird: statt lazy wird eager / applikativ benutzt

Deshalb nennt man applikative Auswertung auch strikte Auswertung (engl. strict evaluation).

#### Applikativartige-Auswertung in Haskell

Haskell wertet lazy aus aber erlaubt, die Auswertungsordnung (zu einem gewissen Grad) zu steuern.

Eine applikativartige Auswertung kann in Haskell mithilfe des Operators ($!) erzwungen werden. Informell wird alles, was rechts vom infixangewandten Operator ($!) steht, ausgewertet, bevor die Auswertung linksseitig vom Operator fortgesetzt wird.

Die Auswertung ist applikativartig, nicht vollständig applikativ, weil in Abhängigkeit des Argumenttyps die Auswertung nur bis zu einer gewissen Tiefe durchgeführt wird. Eingesetzt wird der Operators ($!) hauptsächlich zur Vermeidung exzessiven Speicherverbrauchs.

Späte, faule Auswertung als Standardverfahren:

fac (2\*(3+5))

(E) ->> if (2\*(3+5)) == 0 then 1

else ((2\*(3+5)) \* fac ((2\*(3+5))-1))

...

Teilweise applikative Auswertung:

* Erzwingbar mithilfe des zweistelligen Operators **($!)**

fac $! (2\*(3+5))

(S) ->> fac $! (2\*8)

(S) ->> fac $! 16

(E) ->> if 16 == 0 then 1 else (16 \* fac $! (16-1))

...

Hauptanwendung von ($!) in Haskell zur Minderung des Speicherverbrauchs durch Auswertungsgraphen.

#### Generator/Selektor-Prinzip

Das Generator/Selektor-Prinzip setzt lazy Evaluation als Auswertungsordnung voraus damit die Generator- und Selektor-Ausdrücke ausgewertet werden können. Führt die Auswertung des Generatorausdrucks auf einen konzeptuell nicht endlichen Wert (z.B. Strom statt Liste, nicht endlicher Baum,...), würde die eager Evaluation nicht terminieren, der Selektorausdruck käme nie zum Zuge.

### Auswertungs-Reihenfolgen im Vergleich

**Applikative Auswertungsordnung**

* Fleißige, frühe Argument-Auswertung (engl. applicative, eager order evaluation)
* Wertparameter-, innerste, strikte Auswertung (engl. call-by-value, innermost or strict evaluation)
* Operationalisierung: Linksapplikative, linkestinnerste, frühe, fleißige Auswertung (engl. leftmost- innermost or eager evaluation).
* In ML, Hope, Scheme (ohne Makros)

**Normale Auswertungsordnung**

* Faule, späte Argument-Auswertung (engl. normal order evaluation)
* Namensparameter-, äußerste Auswertung (engl. call-by-name, outermost evaluation)
* Operationalisierung: Linksnormale, linkestäußerste Auswertung (engl. leftmost-outermost evaluation).
* Effiziente Operationalisierung mit Ausdrucksteilung: Späte, faule Auswertung (engl. lazy evaluation), Bedarfsparameter-Auswertung, (engl. call-by-need evaluation)
* in Haskell, Miranda, KRC

**Parameterübergabe-Mechanismen**

Applikative Auswertungsordnung *Call-by-value*

Normale Auswertungsordnung *Call-by-name*

Späte Auswertungsordnung *Call-by-need*

**Argument-Auswertungshäufigkeiten**

Applikative Auswertungsordnung Jedes Argument wird genau einmal ausgewertet.

Normale Auswertungsordnung Jedes Argument wird so oft ausgewertet, wie es benutzt wird.

Späte, faule Auswertungsordnung Jedes Argument wird höchstens einmal ausgewertet.

**Vergleich von Eager und Lazy**

Applikative Auswertung

* einfachere Implementierung
* schnellere Terminierung
* Funktionsargumente werden genau einmal ausgewertet
* Nachteilig ist, dass auch Funktionsargumente ausgewertet werden, die im Funktionsrumpf nicht verwendet werden. Solche Argumente werden unnötig ausgewertet.
* Aus mathematischer Perspektive erscheint fleißige Auswertung oft natürlicher, weil die Undefiniertheit eines Funktionsarguments die Undefiniertheit des Funktionswerts zur Folge hat (Striktheit!), was bei fauler Auswertung i.a. nicht so ist.

Faule Auswertung

vereint die Vorteile applikativer Auswertung (Effizienz) und normaler Auswertung (Terminierungshäufigkeit) benötigt aber mehr Speicher und hat Zusatzaufwand für die Verwaltung geteilter Datenstrukturen.

* Vorteil fauler Auswertung ist, dass die Auswertung so häufig wie nur irgend möglich terminiert.
* Ausdrucke werden nur ausgewertet, wenn ihr Wert zum Wert des Gesamtausdrucks beiträgt, und dann nur genau einmal.
* Schwierigere Implementierung, für expression sharing, um das unnötige Mehrfachauswerten von Ausdrucken zu vermeiden.

Vor und Nachteile ab Folie 1147.

## Programmierparadigmen im Vergleich

**Paradigma**

bestimmte Denkweise, Art der Weltanschauung.

**Programmierparadigma**

Ein Weg Programmiersprachen nach ihren Eigenschaften und Stil, der beim Programmieren praktiziert wird einzuteilen. Diese Eigenschaften sind:

* Dahinterliegendes Berechnungsmodell
* Struktur von Kontrollfluss (Programmablauf) und Datenfluss
* Problemlösung durch Aufteilung in überschaubareren Teilen

Grobe Unterscheidung:

**Imperativ**

auf Maschinenbefehlen aufbauend, Befehlsorientiert

* *Prozedurales Paradigma*
* *Objektorientiertes Paradigma*

**Deklarativ**

auf formalen Modellen / Gleichungen beruhend, man deklariert das erwünschte Resultat, Ergebnisorientiert

* *Funktionales Paradigma*
* *Logikorientiertes Paradigma*

**Prozedurale Programmierung (Paradigma):**

Algorithmus in überschaubare Teile (Je nach Sprache: Unterprogramm, Routine, Prozedur oder Funktion genannt) zerlegen.

Praktisch in allen imperativen Sprachen.

Manche sehen es als Gegenstück zum objektorientierten Paradigma, weil keine Verkapselung von Prozeduren und Variablen.

**Objektorientierte Programmierung (Paradigma):**

Programmierung mit abstrakten Datentypen, behandelt Daten und Funktionen als eine Einheit -> Objekte als First-Class-Entities

**Strukturierte Programmierung (Paradigma):**

Erweitert das Paradigma der prozeduralen Programmierung. Verlangt die Beschränkung auf genau drei Kontrollstrukturen.

**Applikative Programmierung (Paradigma)**

Programmierung mit Ausdrücken und Funktionen die *elementare Werte* (Zahlen, Zeichen, Wahrheitswerte,...) als Argument und Resultat nehmen.

Funktionen sind wichtigstes Abstraktions- und Ausdrucksmittel in applikativer und funktionaler Programmierung.

**Funktionale Programmierung (Paradigma):**

Funktionen als First-Class-Entities ermöglichen Funktionen höherer Ordnung.

Dadurch Rechnen mit Funktionen (Funktionen als Argument und Resultat) möglich (im Gegensatz zur applikativen Programmierung).

* beruhen auf dem mathematischen Funktionsbegriff.
* Funktionale Programme bestehen nur aus Funktionen. Sie sind Systeme rekursiver Funktionsvorschriften, die über Gleichungen, Fallunterscheidungen und Rekursion definiert sind und auf (strukturierten) Daten arbeiten. Die einfachsten Funktionen sind primitive Funktionen. Aus Funktionen werden mithilfe von Funktionen höherer Ordnung neue Funktionen gebildet:

Hauptprogramm -> Hauptfunktion

Programmeingabe, Programmausgabe -> Argument und Resultat der Hauptfunktion

Programmaufbau mit Routinen-> Hauptfunktion ruft andere Funktionen, dadurch Komposition von Funktionen

* bieten effiziente Auswertungsstrategien (lazy), die auch die Arbeit mit unendlichen Strukturen unterstützen.

Funktionale Sprachen

Lisp, ML, SML, Hope, Miranda, OPAL, Haskell, Gofer

Das Besondere an funktionalen Sprachen:

* Funktionskomposition h(x) = (f◦g)(x) = f(g(x))
* Funktionen höherer Ordnung -> Dienen zur Abstraktion
* Höhere Zuverlässigkeit da Korrektheitsüberlegungen und Beweise einfacher sind.
* Effizienz in Programmierung von komplexen Algorithmen
* einfachere Parallelisierbarkeit
* Automatische Listengenerierung, Listenkomprehension

Der Nachteil:

* Schlechte Wartbarkeit
* Keine GUI Programmierung möglich weil keine Zustandsänderungen möglich sind

**Logikorientierte Programmierung (Paradigma):**

logische Aussagen die verbunden werden - Prädikatenlogik.

* Programme sind Systeme von Prädikaten, die durch eingeschränkte Formen logischer Formeln, z.B. Horn-Formeln (Implikation), definiert sind.
* bieten Nichtdeterminismus und Prädikate mit mehreren Eingabe-/Ausgabemodi zur Wiederverwendung von Code.

**Entwicklung von Sprachen**

Bisher schrittweise Abstraktionen mit dem Ziel, Einzelheiten der zugrundeliegenden Rechenmaschine (von Neumann Architektur) zu verbergen.

* Prozeduale Sprachen Teilen Assembler Befehle in überschaubare Teile ein.
* OOP Sprachen führen Sichtbarkeitsebenen und Kapselungen ein, um die Datendarstellung und Speicherverwaltung zu verbergen.

-> Programmformulierung mit Blick auf eine Maschine

* Deklarative Programme verbergen die Auswertungsreihenfolge. Reine deklarative Sprachen verzichten auch auf Zuweisungen, um Seiteneffekte auszuschließen. (Nicht alle deklarativen Sprachen Seiteneffektfrei.)

-> Programmformulierung auf abstraktem, mathematisch geprägten Niveau, ohne eine Maschine im Blick

### Funktionale vs. Imperative Programmierung

**> Imperative Sprachen**

Befehlsbasiertes Programmieren bedeutet: Zustandsänderungen

Programm: Menge von Befehlen (oder Instruktionen, Anweisungen) strukturiert durch Kontrollflussanweisungen

* + Programm ist Arbeitsanweisung für eine Maschine: Anweisungen und Ausdrücke
  + Programmausführung ist die Abarbeitung von Anweisungen, Kontrollfluss (Ausführungsreihenfolge) wird gesteuert (außer bei Ausdrücken)
  + versteckte Seiteneffekte, Programme sind zustands- und ‘zeitbehaftet’

Frage an Programm: In welchem Zustand terminiert Programm angesetzt auf einen Anfangszustand, oder: Was sind die Werte der Variablen nach Terminierung?

Bedeutung des Programms ist die Beziehung zwischen Anfangs- und Endzuständen, die bewirkte Zustandsänderung.

**Anweisungen**

Anweisungen bewirken Zustandsänderungen (Seiteneffekte).

**Wertzuweisungen**

Variablen sind Namen für Speicherplätze, können beliebig oft geändert werden. Namen werden in der zeitlichen Abfolge durch Zuweisungen temporär mit Werten belegt. Namen können durch wiederholte Zuweisungen beliebig oft mit neuen Werten belegt werden (in Schleife oder Rekursion).

**> Funktionale Sprachen**

Gleichungsbasiertes Programmieren bedeutet: Werteberechnungen, Wir beschreiben was wir wollen, nicht wie wir es wollen.

Programm: Menge von Vereinbarungen von Wertegleichheiten von Ausdrücken.

* + Programm ist Ein-/Ausgaberelation: nur Ausdrücke
  + Programmausführung ist Auswertung von Ausdrücken, Keine Kontrollflussspezifikation. Allein Datenabhängigkeiten steuern die Auswertungsreihenfolge, sonst steht sie nicht fest.
  + referentiell transparent, Programme sind zustandsfrei und ‘zeitlos’.

Frage an Programm: Welchen Wert hat ein Ausdruck für konkrete Werte seiner Operanden, wenn er mit den im Programm definierten Wertegleichheiten von Ausdrücken ausgewertet wird?

Bedeutung des Programms ist die Beziehung zwischen Aufrufargumenten von Ausdrücken und ihrem Wert.

**Ausdrücke**

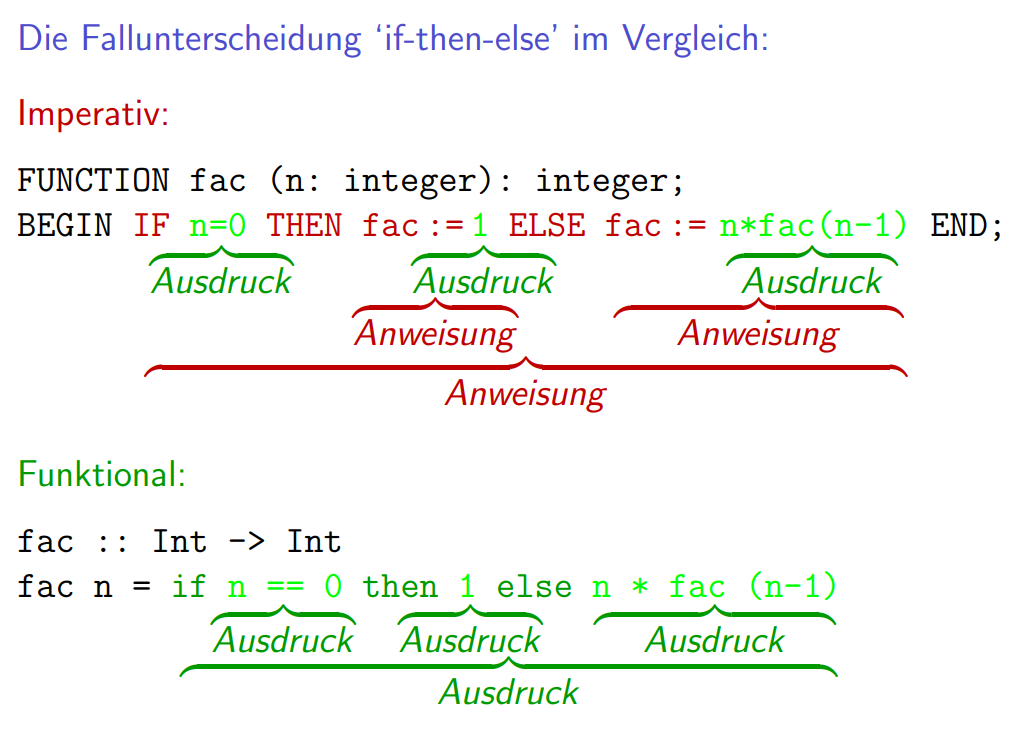
Ausdrücke können ausgewertet werden. Das Resultat ist der Wert eines bestimmten Typs.

keine Zustandsänderungen, keine Seiteneffekte -> referentiell transparent

**Wertvereinbarungen**

Variablen sind Namen für Ausdrücke: Namen werden durch Wertvereinbarungen genau einmal für immer an einen Wert gebunden. (Man spricht damit aber keine Speicherzelle direkt an.) Ihr Wert ist der Wert des Ausdrucks, den sie bezeichnen. Späteres Ändern, Überschreiben oder Neubelegen ist nicht möglich.

**Beispiel: Verzweigung als Kontrollstruktur**

Imperative und funktionale Fallunterscheidung (if-else) sind nur äußerlich sehr ähnlich, konzeptuell aber sehr verschieden. 

### Referentielle Transparenz

**Seiteneffekte**

Auch Wirkung genannt, bewirkt Veränderung eines Zustandes (Variablenwert).

Jeder Programmfortschritt wird über Seiteneffekte erzielt (zB Variablenzuweisungen, Ein- und Ausgaben, Zustandsänderungen).

**Versteckter Seiteneffekt**

Gefährlich, versucht man zu vermeiden.

Beispiel:

f() sortiert ein Array und sammelt statistische Daten. Die sequentielle Ausführung f(x);f(y) wäre kein Problem,

aber f(x);f(x);f(y) würde Array doppelt sortieren und statistischen Daten fälschen.

**Umgang mit versteckten Seiteneffekten**

Imperative Paradigmen versuchen versteckte Seiteneffekte möglichst reduzieren.

Deklarative Paradigmen sind referentiell Transparent:

Vollständige Seiteneffektfreiheit: es werden keine Variablen (Zustandslos) und keine Wertezuweisungen (Seiteneffektfrei) verwendet.

**Definition: Referenzielle Transparenz**

Wenn der Ausdruck durch seinen Wert ersetzt werden kann, ohne die Semantik des Programms dadurch zu ändern.

* 3 + 4 darf mit 7 vertauscht werden

Wenn der Wert des Ausdrucks überall im Programm gleich (hängt von Umgebung ab) und nicht vom Zeitpunkt oder einer bestimmten Reihenfolge der Auswertung. Das ist nur möglich wenn f seiteneffektfrei ist und nicht von Variablen abhängt deren Werte sich im Laufe der Zeit ändern könnten.

* f(x) + f(x) darf mit 2 \* f(x) vertauscht werden (obwohl Bedeutung der Funktion unbekannt ist)

Funktionen in imperativen Paradigmen können referentiell Transparent sein, müssen es aber nicht.

**Umgang mit I/O bei referentieller Transparenz**

Problem: Wenn eine Ausdruck keine Zustände ändern kann, dann kann man nicht herausfinden was es berechnet hat.

Ein- und Ausgaben sind Seiteneffekte, die vollständige referentielle Transparenz unmöglich machen.

Sie werden benötigt, aber sind nicht erlaubt.

In funktionalen Sprachen erlaubt man Ein- und Ausgaben nur gut sichtbar ganz oben in der Aufrufhierarchie (für die erste aufgerufene Funktion die das End-Ergebnis zurückliefert) und verbannt sie aus allen anderen Funktionen.

Ausgabe auf Konsole wird nicht als Seiteneffekt gezählt.

**Imperativ**

Ein-/Ausgabe prinzipiell an jeder Programmstelle möglich.

**Funktional (hier in Haskell)**

Ein-/Ausgabe an bestimmten Programmstellen konzentriert (in meist wenigen global definierten Funktionen der ‘E/A- Schale’).

## Funktionen höherer Ordnung / Funktionale

Funktionen höherer Ordnung sind spezielle Funktionen (die uncurryfiziert sind). Dabei können Funktionale sowohl Resultate sein als auch Argumente.

Funktionen höherer Ordnung kommen auch in der Mathematik vor: zB die Ableitungsfunktion, die Integralrechnung usw.

Vorteile:

Quasi Schablonen von Programmteilen schreiben, die dann durch Übergabe von Funktionen (zum Füllen der Lücken in den Schablonen) ausführbar werden.

* Wiederverwendung von Programmcode.
* Kürzere und meist einfacher zu verstehende Programme.
* Einfachere Herleitung, einfacherer Beweis von Programmeigenschaften (Stichwort: Programmverifikation).
* Besonders nützliche vordefinierte Funktionen auf Listen.

Da Haskell-Funktionen grundsätzlich einstellig sind und damit in den meisten Fällen eine Funktion als Resultat liefern, sind in Haskell die allermeisten Funktionen Funktionen höherer Ordnung. (wenn curryfiziert)

### Abstraktionsprinzipien

Kennzeichnendes Strukturierungsprinzip für…

**Prozedurale Sprachen: Prozedurale Abstraktion**

Operanden werden zu Parametern von Prozeduren / Methoden.

**Funktionale Sprachen: Funktionale Abstraktion**

* + 1. Stufe: Funktionen

Nichtfunktionale Operanden / Werte werden zu Parametern von Funktionen (Gleich wie prozedurale Abstraktion).

Bezeichnet das Ersetzen eines Ausdrucks (oder einer Menge strukturell gleicher Ausdrucke) durch eine Funktion und entsprechende Funktionsaufrufe. Der Ausdruck wird dabei zum Rumpf der Funktion, die Operanden des Ausdrucks zu Parametern der Funktion.

Die Ausdrucksstruktur, d.h. die Verknüpfungsvorschrift der Operanden des Ausdrucks, wird dadurch wiederverwendbar, indem die Funktion für verschiedene Argumente aufgerufen werden kann.

Bewirkt: Wiederverwendung und dadurch kürzerer, verlässlicherer, wartungsfreundlicherer Code.

Beispiel: Statt viele strukturell gleiche Ausdrücke wiederzuverwenden:

(5 \* 37 + 13) \* (37 + 5 \* 13)

(15 \* 7 + 12) \* (7 + 15 \* 12)

(25 \* 3 + 10) \* (3 + 25 \* 10)

...

Eine Funktion, die die Operanden des Ausdrucksmusters als Parameter erhält und mit den ursprünglichen Ausdrucksoperanden(werten) aufgerufen wird.

f :: (Int,Int,Int) -> Int

f (a,b,c) = (a \* b + c) \* (b + a \* c)

Gewinn: Wiederverwendung der gemeinsamen Berechnungsvorschrift durch funktionale Abstraktion 1. Stufe.

* + (Höherer) 2., 3.,... Stufe: Funktionen höherer Ordnung

Verknüpfungsvorschriften werden zu funktionalen Parametern von Funktionen höherer Ordnung.

Bezeichnet zusätzlich zum Herausziehen der Argumente auch das Herausziehen der Struktur möglicher Ausdrucke über diesen Argumenten, d.h. möglicher Verknüpfungsvorschriften der Argumente. Die Funktion wird so zu einer Funktion höherer Ordnung.

Bewirkt: erhöhte die Ausdruckskraft, verbesserte Wiederverwendung. Charakteristisch für funktionale Programmierung.

fac n

| n==0 = 1

| n>0 = n \* fac (n-1)

natSum n

| n==0 = 0

| n>0 = n + natSum (n-1)

natQuSum n

| n==0 = 0

| n>0 = n\*n + natQuSum (n-1)

Die Definitionen von folgen demselben Rekursionsschema:

* Basisfall: eines Basiswertes
* Rekursionsfall: einer Verknüpfungsvorschrift des Argumentwerts n und des Funktionswerts für (n-1)

Diese Gemeinsamkeit legt es nahe: Rekursionsschema, Verknüpfungsvorschrift, Basiswert herauszuziehen, zu abstrahieren. Eine Abstraktion höherer Stufe:

rekSchema :: Int -> **(Int -> Int -> Int)** -> Int -> Int

rekSchema basiswert verknuepfe n

| n==0 = basiswert

| n>0 = verknuepfe n (rekSchema basiswert verknuepfe (n-1))

Uncurryfizierte Signatur:

rekSchema :: (Int,**(Int -> Int -> Int)**,Int) -> Int

Dadurch können wir alle 3 Funktionen mittels einer Funktion ausdrücken die nur den Verknüfungsoperator verlangt.

fac n = rekSchema 1 (\*) n

natSum n = rekSchema 0 (+) n

natQuSum n = rekSchema 0 (\x y -> x\*x + y) n

Gewinn: Wiederverwendung des gemeinsamen Strukturmusters der Funktionen fac, natSum und natQuSum durch funktionale Abstraktion höherer Stufe.

### Curryfiziert vs. Uncurryfiziert

Entscheidend für die Unterscheidung ist die Anzahl der konsumierten Argumente

**Einzeln Argument für Argument: curryfizierte Funktion f x, f x y, f x y z,...**

Nur curryfizierte Funktionen unterstützen das Prinzip partieller Auswertung und damit das Prinzip: Funktionen liefern Funktionen als Ergebnis!

**Alle auf einmal als Tupel: uncurryfizierte Funktion f(x), f(x,y), f(x,y,z),...**

Mathematische Definition mit Produkt/Tupelbildung in einer Funktion mit mehreren Eingnagswerten wird in Haskell übersetzt. (Da Funktionen in Haskell jeweils nur ein Argument konsumieren können).

Mathematische Definition

f: (N × N × N × N) 🡪 N

wird in Haskell zu

function :: (Int, Int, Int, Int) -> Int

Uncurryfiziert gleiche Funktionalität wie curryfizierte Funktion aber nicht so flexibel.

Faustregel: curryfiziert, wo möglich, uncurryfiziert nur dort, wo nötig.

Beispiel: Curryfizierte Deklaration

binom :: Integer -> (Integer -> Integer)

binom n k = div (fac n) (fac k \* fac (n-k))

Man kann auch die Klammer auslassen und salopp behaupten binom ist eine zweistellige Funktion (obwohl alle Funktionen nur 1 Argument konsumieren können).

Beispiel: Uncurryfizierte Deklaration

binom :: (Integer,Integer) -> Integer

binom (n,k) = div (fac n) (fac k \* fac (n-k))

**curry und uncurry in Haskell**

curry :: ((a,b) -> c) -> (a -> b -> c)

curry f x y = f (x,y)

uncurry :: (a -> b -> c) -> ((a,b) -> c)

uncurry g (x,y) = g x y

### Operatorabschnitte (engl. operator sections) in Funktionalen

Partiell ausgewertete Binäroperatoren in Haskell.

Funktionen in Haskell haben eine Stelligkeit von 1: Sie können bei jedem Auswertungsschritt nur ein Argument aufnehmen.

Ein Operatorabschnitt ist eine partiell ausgewertete Funktion, die daher nur noch mehr Argumente benötigt, um ein nicht-funktionales Ergebnis zu liefern.

Kommutative Binäroperationen:

(\*2) gleichzusetzen mit (\x -> x\*2)

(2\*) gleichzusetzen mit (\x -> 2\*x)

Nicht-Kommutative Binäroperationen:

(`div` 2) gleichzusetzen mit (\x -> div x 2)

(2 `div`) gleichzusetzen mit (\x -> div 2 x)

(binom 45) gleichzusetzen mit (\x -> binom 45 x)

Verallgemeinert:

(op) = (\x -> (\y -> x `op` y))

(x `op`) = (\y -> x `op` y)

(`op` y) = (\x -> x `op` y)

**Bildung konstanter Funktionen (engl. λ-Lifting)**

Als λ-Lifting bezeichnet man die Konstruktion einer konstanten Funktion zu einem gegebenen Wert w, die diesen Wert als einzigen Funktionswert hat (und deshalb eine konstant(wertige) Funktion ist.

In 4 verschiedenen syntaktischen Varianten:

lifting :: a -> (b -> a)

**lifting x = \\_ -> x**

lifting x = g where g y = x

lifting x = g where g \_ = x

lifting x = \y -> x

**Punktfreiheit**

Eine punktfreie Funktionsdeklaration verzichtet auf die Argumente bei der Definition des Funktionsrumpfes, eine nichtpunktfreie nennt die Argumente ausdrücklich.

Beispiel anhand der Fibonacci-Funktion:

Nichtpunktfreie (oder: argumentbehaftete) Deklaration von fib:

type Nat0 = Integer

fib :: Nat0 -> Nat0

fib n = if n <= 1 then n else fib (n-2) + fib (n-1)

Punktfreie (oder: argumentlose) Deklaration von s:

fib’ :: Integer -> Integer

fib’ = \n -> if n <= 1 then n else fib’ (n-2) + fib’ (n-1)

fib’’ :: Integer -> Integer

fib’’ = fib

## Typisierung

**Typsysteme**

Das Typsystem ist der Teil der Programmiersprache welcher die Typisierung (die Überprüfung) durchführt.

Typsysteme sind logische Systeme, die uns erlauben, Aussagen der Form *‘exp ist Ausdruck vom Typ t’* zu formalisieren und sie mithilfe von Axiomen und Regeln des Typsystems zu beweisen.

**Typisierung**

Ziel der Typisierung ist die Vermeidung von Typverletzungen / Erhaltung der Typkonsistenz in der Sprache.

Sprachen die ein Typsystem beinhalten nennt man typisiert.

Fast allen aktuellen Programmiersprachen sind typisiert um, Zuweisungsfehler zu vermeiden, Typkonsistenz zu sichern.

**Typkonsistenz und Typverletzungen**

Typen sind miteinander konsistent, wenn die Typen der Operanden mit der Operation zusammenpassen (Die Operation auf diesen Typen definiert ist), andernfalls tritt ein Typfehler auf.

### Typisierung von Haskell

Starke, statische Typisierung mit Typinferenz (es lohnt sich aber trotzdem zu typisieren da dadurch die Qualität der Fehlermeldungen steigt). Gültige Ausdrücke haben wohldefinierte Typen und heißen wohlgetypt:

Typen können explizit angegeben sein, müssen sie aber nicht da sie inferiert werden.

### Einteiltung der Typisierungsarten

Strongly vs. Weakly / loosly stark vs. schwach

Static vs. Dynamic Kompilierzeit vs. Laufzeit

Manifest vs. Inferred explizit vs. implizit mit Typinferenz

Nominal vs. Structural nominal vs. strukturell

**Starke / Schwache Typisierung**  
Abhängig von der Typsicherheit einer Sprache.

Wie streng unterscheidet die Sprache die Typen? Welche Datentypen können ineinander umgewandelt werden? Erlaubt sie implizite Typumwandlungen? Erlaubt sie unsichere Typumwandlungen, bei denen z.B. Werte verloren gehen können?

Typfehler werden in

* schwach getypten Sprachen erst zur Laufzeit
* stark getypten Sprachen bereits zur Übersetzungszeit erkannt.

**Statische / Dynamische Typisierung**  
Typprüfungen können zur Übersetzungszeit oder zur Laufzeit vorgenommen werden. Durch Auswertung jeder Art von Kontextinformation in Ausdrücken, Funktionsdefinitionen und Typklassen

verwendete Operatoren ((+), (/), (&&), (++)), Konstanten (2, 3.57, True, (2,3), [], [2,3,4],...), Muster ((‘c‘:cs), (x,False),...), etc.

Beeinflusst Zeitpunkt an dem Fehler entdeckt werden.

Wenn Typ einer Variable bekannt: klar, welche Werte in ihr enthalten sein können - Wissen statt Spekulation; frühe Entscheidung, welcher Typ verwendet werden soll, erleichtert Planbarkeit und erhöht Verständlichkeit des Programms, schränkt aber gleichzeitig Möglichkeiten/Flexibilität ein.

Statische Typisierung

Vorteile:

* Verlässkichkeit Weniger Typfehler zur Laufzeit, verbesserte Zuverlässigkeit, Fehler zeigen sich früher
* Leichtere Lesbarkeit da Information von statischen Typen zuverlässig (wenn explizit deklariert)
* Typinferenz auch möglich
* weniger Fallunterscheidungen im Compiler da Typ früh bestimmt, verbessert Planbarkeit für Compiler und Verständlichkeit für Mensch
* Erhöhte Effizienz: Mehr Informationen für Programmoptimierung zur Übersetzungszeit

Nachteile:

* Geringere Flexibilität zur Laufzeit. Benötigt Typumwandlungen die unsicher sein können.
* Man kann nicht alles statisch herleiten zb Array-Grenzen (aber manche Sprachen kommen ausschließlich mit statischer Typprüfung aus, zb Haskell). Schränkt Sprache aber ein und kann effektiv zu mehr Aufwand führen.
* Dynamische Sprachen möglicherweise sogar sicherer, weil diese besser getestet werden (bei Suche nach Typfehlern werden nebenbei auch andere Fehler erkannt)

Dynamische Typisierung

Vorteile:

* optionale Typdeklarationen möglich oder Typ in den Variablennamen schreiben

**Explizite / Implizite Typisierung**  
Datentyp kann explizit genannt werden oder per Typableitung (*type inference*) ermittelt werden -> wenn Typen im Quellcode nicht explizit hingeschrieben werden, aber der Compiler diese im Rahmen der statischen Typprüfung aus der Programmstruktur ableiten kann.

Implizite Typisierung / Typisierung

Vorteile:

* Typinferenz spart das Anschreiben von Typen.

Nachteile:

* Zur Verbesserung der Lesbarkeit kann und soll man Typen explizit anschreiben -> Explizite Typisierung verbessert Lesbarkeit.
* Typinferenz und Ersetzbarkeit durch Untertypen verträgt sich nicht (im selben Teil des Programms).

### Typprüfung

**Muster-Prüfung / Musterkonsistenz**

* Eine Variable als Muster ist mit jedem Typ konsistent.
* Ein Literal oder Konstante als Muster ist mit ihrem Typ konsistent.

Beispiele:

* Das Muster (42:xs) ist konsistent mit dem Typ [Int].
* Das Muster (x:xs) ist konsistent mit jedem Listentyp**.**

**Monomorphe Typprüfung**

Typprüfung stellt fest: Ein Ausdruck ist entweder

* wohlgetypt u. hat einen eindeutig bestimmten konkreten Typ.
* nicht wohlgetypt und hat überhaupt keinen Typ.

**Polymorphe Typprüfung (mit Typklassen)**

Dreistufiger Prozess aus:

1. Unifikation

2. Analyse (einschl. Instanz- und Typklassendeklarationen)

3. Simplifikation

Typprüfung stellt durch das Lösen von Typkontextsystemen (engl. constraint satisfaction) unter Unifikation von Typausdrücken fest:

Ein Ausdruck ist

* wohlgetypt und hat einen, mehrere, möglicherweise unendlich viele konkrete Typen.
* nicht wohlgetypt und hat überhaupt keinen Typ.

Beispiel: Die polymorphe Funktionssignatur

length :: [a] -> Int

mit beliebigem monomorpher Typ

[Int] -> Int

[(Bool,Char)] -> Int

[String -> String] -> Int

[Bool -> Bool -> Bool] -> Int

in Aufrufkontexten wie:

length [1,2,3], length [True,False,True], length [], length [(+),(\*),(-)]

lässt sich ein monomorpher Typ eindeutig erschließen:

length :: [Int] -> Int

Beachte: Der Kontext length [] erlaubt nur auf [a] -> Int für den Typ zu schließen.

Beispiel:

g (m,zs) = m + length zs

g bezeichnet eine Funktion, die als Argument Paare erwartet, an deren Komponenten folgende Bedingungen gestellt sind:

* 1-te Komponente: m muss von einem numerischen Typ sein, da m als Operand von (+) verwendet wird.
* 2-te Komponente: zs muss vom Typ [b] sein, da zs als Argument der Funktion length verwendet wird, die den

Typ ([b] -> Int) hat.

Beides zusammen erlaubt den allgemeinsten Typ von g zu erschließen: g :: (Int, [b]) -> Int

Beispiel: Funktions-Komposition

g :: (Int, [b]) -> Int

g (m,zs) = m + length zs

f :: (a,Char) -> (a,[Char])

f (x,y) = (x,[‘a‘ .. y])

Die Funktions-Komposition (g . f)

* In der Mathematik: (g . f) = g(f(x))
* Das Resultat von f ist das Argument von g.

Das Resultat von f ist vom Typ (a,[Char]).

Das Argument von g ist vom Typ (Int,[b]).

* Dadurch:

(g . f) :: (a,Char) -> (a,[Char]) wobei (Int, [b]) verlangt wird -> Int

Damit sind noch zu bestimmen: Die allgemeinst möglichen Typen für die Typvariablen a und b die die obigen Bedingungen erfüllen. -> Der Schlüssel dafür: Unifikation.

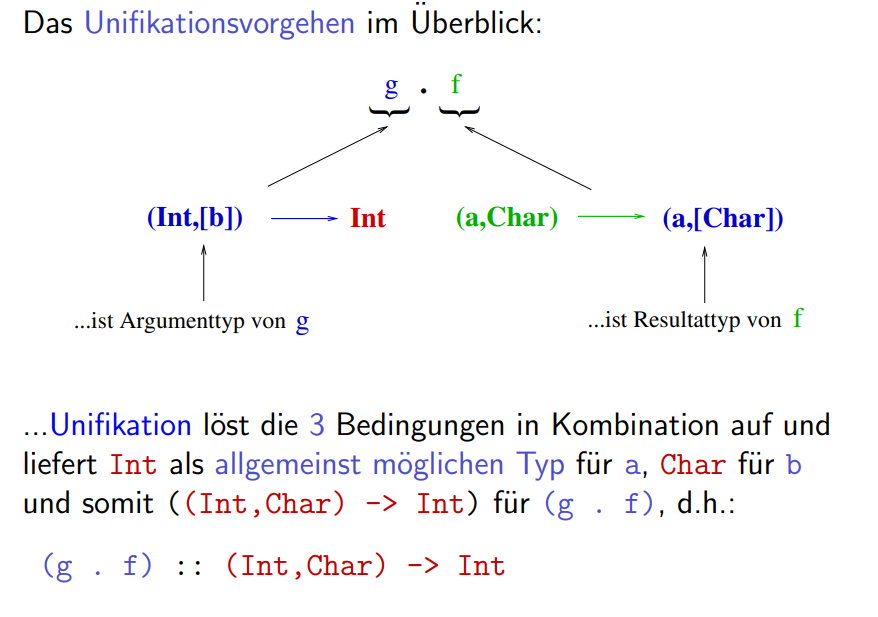
#### Unifikation

Um den allgemeinsten, gemeinsamen Typen für 2 Variablen zu bestimmen.

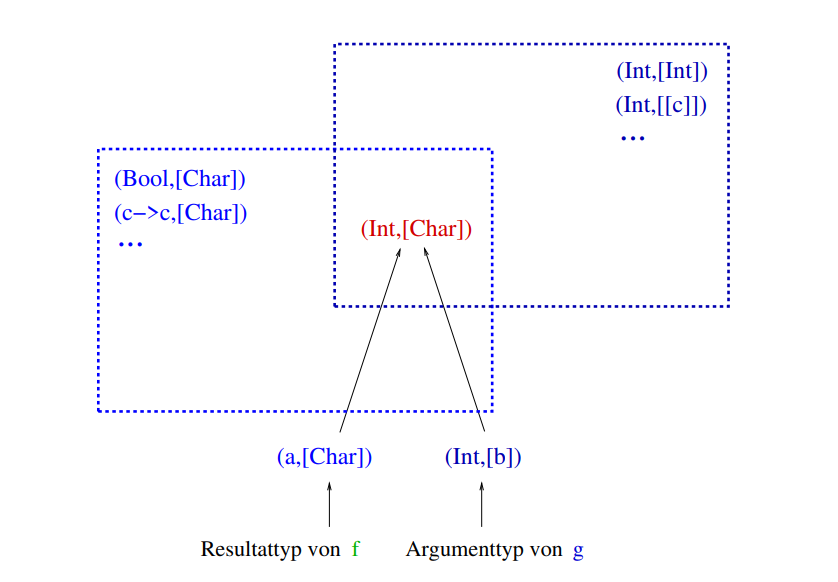
(g . f)

f :: (a,Char) -> (a,[Char])

g :: (Int, [b]) -> Int



Suche: allgemeinste gemeinsame Instanz einer Menge



**Unifikation, Unifikationsaufgabe**

* ist die Bestimmung der allgemeinsten gemeinsamen (Typ-) Instanz (engl. most general common (type) instance) einer Menge von Typausdrücken und der zugehörigen Substitution.
* Konstanten und Variablen werden in Haskell bei Unifikation unterschiedlich behandelt.
* bestimmt allgemeinstmögliche mehrere Typbedingungen zugleich erfüllende Typausdrücke, die allgemeinste gemeinsame Instanz einer Menge von Typausdrücken sind.
* wertet dafür Kontextbedingungen in Kombination aus.
* führt i.a. zu polymorphen Typausdrücken.
* kann fehlschlagen.

## Typen

In funktionalen Sprachen sind Variablen Namen für Ausdrücke. Ein Ausdruck kann ein elementarer Wert mit einem Datentyp sein oder eine Funktion.

Jede (Typ-) Variable hat einen Typ als Wert:

* Funktionale Typen deklarierbar mit syntaktischer Funktions-Signatur
* elementare Datentypen mit vordefinierten oder selbstdefinierten Deklarationen (wie newtype, data)

**Ersetzbarkeitsprinzip**

Ein Typausdruck b ist Instanz eines Typausdrucks a [a <- b], wenn a sich zu b spezialisieren lässt; wenn b eine Teilmenge von Typen von a beschreibt.

### Datentyp-Deklarationen in Haskell

mittels drei Sprachkonstrukten: type, newtype und data

#### type: Typsynonyme

Typnamen, Typaliase, Synonyme für bereits existente Typen.

Erhöhen die Typsicherheit bzw. ungeeignete Verwendung nicht.

**type** Kurs = Float

**type** Pegelstand = Float

**type** Koordinate = Float

#### newtype: Neue Typen

Statt Typsynonyme, um Typsicherheit und Performance zu erhöhen. Die Nutz-Daten, liegen jetzt geschützt hinter Datenkonstruktoren.

**newtype** Kurs = **K** Float

**newtype** Pegelstand = **Pgl** Float

**newtype** Koordinate = **Koordinate** Float

Wichtig: *Datenwert-Konstruktoren* können als Funktionsdefinitionen gelesen werden.

K, Pgl, Koordinate sind Datenkonstruktor-Namen und der Wert danach sind Nutzdatentypen.

Beschränkt auf 1 Konstruktor mit 1 Datenfeld.

#### data: Algebraische Datentypdeklaration

Algebraische Typen mit data-Deklaration erlauben neue Datentypen mit beliebig strukturierten Werten einzuführen.

Mehrere Konstruktoren und mehrere Datenfelder erlaubt.

Anders als type-Deklarationen gewähren newtype- und data-Deklarationen Typsicherheit. In

Es kann jede newtype-Deklaration durch eine data-Deklaration ersetzt werden, aber nicht umgekehrt (siehe Summentypen).

Allerdings ist damit ein Performanzverlust verbunden, da (anders als bei newtype) bei data-Deklarationen die Datenwertkonstruktoren nicht nur Übersetzungszeit, sondern auch zur Laufzeit einen gewissen Berechnungsaufwand verursachen.

(Übersicht siehe Folie 523)

* **Aufzählungstypen** / Enumerations

Mehrere 0-Stellige Daten-Konstruktoren

Eigentlich ein Spezialfall von Summentypen

data Jahreszeiten = Fruehling | Sommer | Herbst | Winter

data Geschlecht = Maennlich | Weiblich

* **Produkttypen** / Verbundtypen, Record-Typen

Ein mehrstelliger Daten-Konstruktor

Produkttyp data Person = **P** Vorname Nachname Geschlecht

Unechter Produkttyp data Person = **P** (Vorname,Nachname,Geschlecht)

Tupeltyp **newtype** Person = **P** (Vorname,Nachname,Geschlecht)

Tupeltyp **type** Person = (Vorname, Nachname, Geschlecht)

Vergleich mit newtype: Produkttyp vs. Tupeltyp

newtype erlaubt Tupeltypen (entsprechend Produkttypen mit einem Datenfeld – unechte Tupeltypen mit nur einem Datenfeld) eine neue Identität zu verleihen.

Typsicherheit: Beides gewährleistet Typsicherheit.

Performance: Vorteil von newtype Tupeltyp ist aber, dass der Daten-Konstruktor des Produkttyps nur zur Schreibzeit und zur statischen Typprüfung verwendet wird. Für data-Typen ist dies nicht möglich, weil es mehr als einen (Daten-) Konstruktor gibt. Produkttypen beanspruchen also zur Laufzeit zusätzliche Kapazitäten.

Fehlermeldungen-Qualität: bei Produkttypen besser

* **Summentypen** / Variantentypen, Vereinigungstypen

ein oder mehrere null-, ein- oder mehrstellige (Daten-) Konstruktoren.

Typen mit Werten, die sich aus der Vereinigung der Werte verschiedener (auch algebraischen Datentypen) zusammensetzen.

##### Unterschied zwischen Summentypen und Produkttypen

**Summentypen mögliche Werte**

1 + 1 + 1

data Foo = Foo | Bar | Baz -- 3 Werte

1 + 1 + 2

data Foo1 = Foo1 | Bar1 | Baz1 Bool -- 4 Werte

2 + 2 + 2

data Foo2 = Foo2 Bool | Bar2 Bool | Baz2 Bool -- 6 Werte

**Produkttypen**

1

data Foo3 = Foo3 -- 1 Wert

2\*\*1 = 2

data Foo4 = Foo4 Bool -- 2 Werte

2\*\*2 = 2 \* 2

data Foo5 = Foo5 Bool Bool -- 4 Werte

2\*\*3 = 2 \* 2 \* 2

data Foo6 = Foo6 Bool Bool Bool -- 8 Werte

**gemischt**

2 \* 2 \* 2 + 2 + 1

data Foo7 = Foo7 Bool Bool Bool | Bar7 Bool | Baz7 -- 11 Werte

##### Rekursive Summentypen

Binärbäume:

data Baum = Leer | Wurzel Baum Int Baum deriving (Eq,Ord,Show)

Trinärbäume:

data TBaum = Nichts | Gabel Person TBaum TBaum TBaum deriving (Eq,Ord,Show)

N-stellige Bäume:

data NBaum = NB Int [NBaum] deriving (Eq,Ord,Show)

data NBaum0 = NB0 (Person,[Anschrift]) [NBaum0] deriving (Eq,Ord,Show)

data Nadelbaum = Nb String Int Char Baum TBaum [Nadelbaum] deriving...

Suchbäume:

type Schluessel = Int type Information = String

data Suchbaum = Sb Schluessel Information | Sk Schluessel Information Suchbaum Suchbaum

deriving (Eq,Ord,Show)

##### Vordefinierte algebraische Datentypen

**Aufzählungstypen**

Typ der Ordnungswerte, 3 Werte:

data Ordering = LT | EQ | GT deriving (Eq,Ord, Bounded, Enum, Read, Show)

Typ der Wahrheitswerte, 2 Werte:

data Bool = False | True deriving (Eq,Ord,Bounded,Enum,Read,Show)

Trivialer Typ (oder *Nulltupeltyp*), 1 Wert:

data () = () deriving (Eq,Ord,Bounded,Enum,Read,Show)

**Summentypen**

Der Möglicherweise-Typ (polymorph)

data Maybe a = Nothing | Just a deriving (Eq,Ord,Read,Show)

Der Entweder/Oder-Typ (polymorph)

data Either a b = Left a | Right b deriving (Eq,Ord,Read, Show)

##### Feldsyntax

Ziel: Transparente, sprechende Typdeklarationen. In Haskell bieten sich dafür drei Möglichkeiten an:

1. Kommentierung

2. Typsynonyme

3. Feldsyntax (Verbundtypsyntax) mit dem Zusatzvorteil

– ‘geschenkter’ Selektorfunktionen

– wesentlich vereinfachte weitere Verarbeitungsfkt.

**1) durch Kommentierung**

newtype Gb = Gb (String,String,String) deriving (Eq,Ord,Show)

data G = M | W deriving (Eq,Ord,Show)

data Meldedaten = Md String -- Vorname

String -- Nachname

Gb -- Geboren (tt,mm,jjjj)

G -- Geschlecht (m/w)

String -- Gemeinde

String -- Strasse

Int -- Hausnummer

Int -- PLZ

String -- Land

deriving (Eq,Ord,Show)

**2) durch Typsynonyme**

type Vorname = String

type Nachname = String

type Ziffernfolge = String

type Zf = Ziffernfolge

newtype Gb = Gb (Zf,Zf,Zf) deriving (Eq,Ord,Show)

type Geboren = Gb

data G = M | W deriving (Eq,Ord,Show)

type Geschlecht = G

type Gemeinde = String

type Strasse = String

type Hausnummer = Int

type PLZ = Int

type Land = String

data Meldedaten = Md Vorname Nachname Geboren Geschlecht Gemeinde Strasse Hausnummer PLZ Land deriving (Eq,Ord,Show)

**3) durch Feldsyntax (oder: Verbundtypsyntax)**

type Ziffernfolge = String

type Zf = Ziffernfolge

data G = M | W deriving (Eq,Ord,Show)

newtype Gb = Gb (Zf,Zf,Zf) deriving (Eq,Ord,Show)

data Meldedaten = Md { vorname :: String,

nachname :: String,

geboren :: Gb

geschlecht :: G,

gemeinde :: String,

strasse :: String,

hausnummer :: Int,

plz :: Int,

land :: String

} deriving (Eq,Ord,Show)

**Typgleiche Felder können in der Feldsyntax durch Beistrich getrennt zusammengefasst werden:**

… { vorname,

nachname,

gemeinde,

strasse,

land :: String,

geboren :: Gb,

geschlecht :: G,

hausnummer,

plz :: Int

} …

**Feldnamen in Alternativen dürfen wiederholt verwendet werden, wenn ihr Typ für alle Vorkommen ident ist**

type Ziffernfolge = String

type Zf = Ziffernfolge

data G = M | W deriving (Eq,Ord,Show)

newtype Gb = Gb (Zf,Zf,Zf) deriving (Eq,Ord,Show)

data Meldedaten = Md { vorname,

nachname,

gemeinde,

strasse,

land :: String,

geboren :: Gb

geschlecht :: G,

hausnummer,

plz :: Int

}

| KurzMd { vorname, nachname :: String } deriving (Eq,Ord,Show)

**Die Feldnamen selbst sind die Selektorfunktionen:**

Folgende Funktionen sind obsolet:

vornameVon :: Meldedaten -> Vorname

vornameVon = vorname

nachnameVon :: Meldedaten -> Nachname

nachnameVon = nachname

...

plzVon :: Meldedaten -> PLZ

plzVon = plz

landVon :: Meldedaten -> Land

landVon = land

**Setter lassen sich wesentlich einfacher schreiben:**

setzeVorname :: Vorname -> Meldedaten -> Meldedaten

setzeVorname vn md = md {vorname = vn}

setzeNachname :: Nachname -> Meldedaten -> Meldedaten

setzeNachname nn md = md {nachname = nn}

...

erzeugeMdMitVorname :: Vorname -> Meldedaten

erzeugeMdMitVorname vn = Md {vorname = vn}

erzeugeMdMitNachname :: Nachname -> Meldedaten

erzeugeMdMitNachname nn = Md {nachname = nn}

...nicht genannte Felder werden automatisch ‘undefiniert’ gesetzt.

Es können auch mehrere Felder gleichzeitig gesetzt werden.

### class: Typklassen (Haskell spezifisch)

Typklassen sind das Sprachmittel von Haskell, Funktionen (genauer: Operator- und Relatorsymbole) zu überladen und typspezifisch zu implementieren.

Typclasses (ähnlich wie interfaces in java) setzten Funktionen fest die ihre Instanzen implementieren sollten.

>> Typvariablen haben Typen als Wert. Typklassen haben Typen als Instanz.

Ist z.B. ein Typ A Element der Typklasse Num, so kann sich der Programmierer darauf verlassen, dass auf A-Werten alle in Num aufgeführten Funktionen definiert sind, aber auch alle Funktionen von denen die Typklasse Num erbt, nämlich Eq und Show.

Zusammengefasst:

* Beinhalten eine beliebige Anzahl an Funktionen die man mit Typen die Instanzen sind ausführen kann.

Durch Instanzbildungen der Typklasse für verschiedene Typen wird die Bedeutung dieser Funktionen typspezifisch.

* Können Protoimplementierungen (engl. default implementations) bereitstellen die überschrieben werden dürfen.
* Typenwerden durch Instanzbildung (Implementierung der notw. Signaturen) zu Instanzen einer Typklasse.

Es sind die typspezifischen Bedeutungen der überladenen Funktionen einander entsprechend, ihre Funktionalität miteinander vergleichbar, jeweils typspezifisch zugeschnitten.

* Können von anderen Typklassen **erben**.
* Können geerbte Implementierungen **überschreiben**.

#### Automatische und manuelle Instanzbildung

**Automatische Typklasseninstanzbildung**

Algebraische und neue Typen (keine Typsynonyme) können mit deriving-Klausel automatisch als Instanzen vordefinierter Typklassen angelegt werden.

Nur von Eq, Ord, Enum, Bounded, Show, Read. Für andere Typklassen, gleich ob vor- oder selbstdefiniert, sind zur Instanzbildung stets instance-Deklarationen erforderlich.

Automatische Instanzbildungen können ausschließlich für die vordefnierten Typklassen Eq, Ord, Enum, Bound, Read und Show vorgenommen werden. Die automatische Instanzbildung nimmt dabei die manuelle Instanzbildung in ‘naheliegendster’ Weise vor. Das erfordert ein Vorwissen der ‘naheliegendsten’ Bedeutung der in einer Klasse vorgesehenen Funktionen. Dieses Vorwissen ist nicht für alle Typklassen für den Compiler ersichtlich. Deshalb ist automatische Instanzbildung nicht für alle Typklassen möglich.

Beispiel: Gleichheit automatisch interpretiert als Gleichheit in Struktur und Benennung.

data (Eq a, Eq b, Eq c) => Baum a b c = Blatt a b | Wurzel (Baum a b c) c (Baum a b c) deriving Eq

ist gleichbedeutend mit:

**Manuelle Typklasseninstanzbildung**

data (Eq a, Eq b, Eq c) => Baum a b c = Blatt a b | Wurzel (Baum a b c) c (Baum a b c)

instance (Eq a, Eq b, Eq c) => Eq (Baum a b c) where

(Blatt u v) == (Blatt x y) = (u == x) && (v == y)

(Wurzel ltb z rtb) == (Wurzel ltb0 z0 rtb0) = (ltb == ltb0) && (z == z0) && (rtb == rtb0)

\_ == \_ = False

Manuelle Typklasseninstanzbildung erlaubt Gleichheit abweichend von ‘offensichtlicher’ Gleichheit, beliebig durch Verwendung der Nutzungsdaten zu implementieren. Es gibt eine Minimalvervollständigung bei Instanzbildungen.

#### Vordefinierte und Selbstdefinierte Typklassen

**Vordefinierte Typklassen**

Beispiel Eq: Für Typen, deren Werte absolut verglichen werden können auf Gleichheit und Ungleichheit

* verlangt von Instanzen die Implementierung von zwei Wahrheitswertfunktionen (oder Prädikaten): (==), (/=).
* stellt für beide Wahrheitswertfunktionen eine Protoimplementierung zur Verfügung:

class Eq a where

(==), (/=) :: a -> a -> Bool

x /= y = not (x==y) -- Protoimplementierung f. (/=)

x == y = not (x/=y) -- Protoimplementierung f. (==)

class **Eq a =>** Ord a where

(<), (<=), (>), (>=) :: a -> a -> Bool

max, min :: a -> a -> a

compare :: a -> a -> Ordering

-- Protoimplementierungen ab hier

compare x y

| x == y = EQ

| x <= y = LT

| otherwise = GT

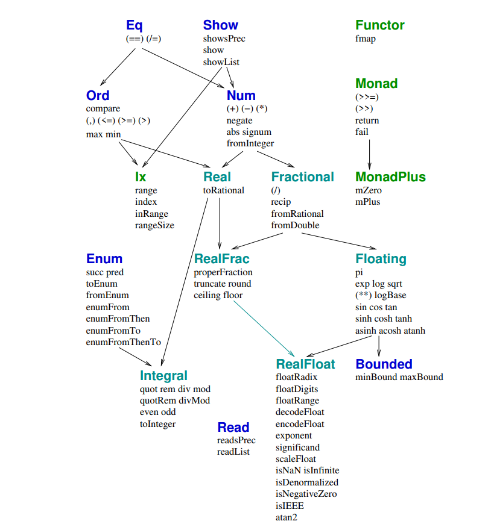
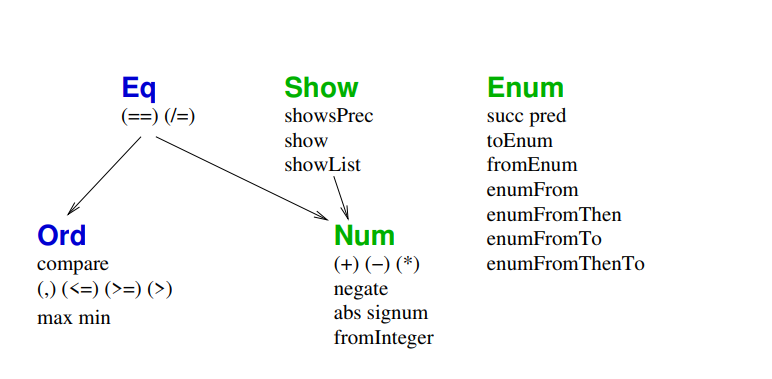
(<=) x y = (x < y) || (x == y)

(<) x y = y < x

...

* Die Typklasse Ord erweitert die Klasse Eq. Jeder Typ, der zu einer Instanz der Typklasse Ord gemacht werden soll, muss bereits Instanz der Typklasse Eq sein.
* Für eine vollständige Instanzbildung reicht es, Implementierungen der Relatoren (==) und (<) anzugeben.

Typklassen in Haskell bilden eine Hierarchie:



Im Überblick:

* *Eq (Gleichheit):* Werte von Eq-Typen müssen auf Gleichheit und Ungleichheit vergleichbar sein.
* *Ord (Vergleichbarkeit):* Werte von Ord-Typen müssen über Gleichheit und Ungleichheit hinaus bezüglich ihrer relativen Größe vergleichbar sein.
* *Show (Ausgabe):* Werte von Show-Typen müssen eine Darstellung als Zeichenreihe besitzen.
* *Read (Eingabe):* Klasse der Typen, deren Werte aus Zeichenreihen herleitbar sind
* *Num (Numerisch):* Werte von Num-Typen müssen mit ausgewählten numerischen Operationen verknüpfbar sein, d.h. addiert, multipliziert, subtrahiert, etc. werden können.
* *Enum (Aufzählung):* Werte von Enum-Typen müssen aufzählbar sein, d.h. einen Vorgänger- und Nachfolgerwert innerhalb des Typs besitzen.

**Mehrfachvererbung**

auf Typklassenebene ist ebenfalls möglich: class (Eq a, Show a) => Num a where

Hierarchie zeigt sich auch in den Signaturen:

class (Num a, Ord a) => **Real a** where...

**Selbstdefinierte Typklassen**

Es bietet sich an Funktionen in einer selbstdefinierten neuen Typklasse zu bündeln, so Namen einzusparen und wiederzuverwenden.

Beispiel:

class (Ord a, Fractional a) => **Analysierbar a** where

**auswertung** :: (a,a) -> (a,a,a)

**reihenausw** :: [(a,a)] -> [(a,a,a)]

**geglaettet** :: a -> a -> a

reihenausw [] = [] -- Protoimplementierung für reihenausw (wird überschrieben)

reihenausw ((x,y) : xys) = (auswertung (x,y)) : reihenausw xys

geglaettet x y = (x+y)/2 -- Vollst. Implementierung für geglaettet

Instanzbildung: Minimalvervollst. bei Instanzbildungen für Analysierbar: Implementierung von auswertung

newtype Kurs = K Float deriving (Eq,Ord,Show)

instance **Analysierbar** Float where

auswertung (f1,f2) = (f1,f2,geglaettet f1 f2)

instance **Analysierbar** Kurs where

auswertung (K k1,K k2) = (K k1,K k2,K (geglaettet k1 k2))

geglaettet k k0 = (4\*k+6\*k0)/10 -- überschrieben!

Erweiterung von Analysierbar mit einer neuen Typklasse Warnung:

class **Analysierbar** a => **Warnung** a where

warnung :: (a,a) -> String

warnreihe :: [(a,a)] -> String

warnreihe xys = warnung (wr xys (0,0))

where wr [] pq = pq wr ((x,y) : xys) (p,q) = wr xys (x+p,y+q)

Instanzbildung: Minimalvervollst. bei Instanzbildungen für Warnung: Implementierung von warnung

instance **Warnung** Kurs where

warnung (K k1,K k2)

| k2 > 9\*k1 = "Verkaufen! Aktie zu spekulativ."

| k2 > 6\*k1 = "Halten! Aktie an Spekulationsschwelle."

| k2 > 3\*k1 = "Zukaufen! Aktie hat Phantasie."

| otherwise = "Verkaufen! Aktie ohne Phantasie."

#### Typklassen in Haskell vs. Objektorienterte Klassen

Haskells Typklassenkonzept unterscheidet sich wesentlich vom Klassenkonzept objektorientierter Sprachen.

**Objektorientiert: Klassen**

* dienen der Strukturierung von Programmen.
* Dienen als Schablone für Objekterstellung

**In Haskell: Typklassen**

* dienen nicht der Strukturierung von Programmen
* sind Sammlungen von Typen, deren Werte typspezifisch, aber mit Funktionen / Relationen / Operatoren gleichen Namens (und ähnlicher Funktionalität) bearbeitet werden sollen ((==),(>=),(+),(-), etc.).
* dienen der Organisation und Verwaltung von Überladung (der Protoimplementierungen) und Erweiterungen der verlangten Funktionen für die Typen die Instanzen der Typklasse sind
* Erlauben einzelne (unecht) polymorphe Funktionen zu schreiben, die sich auf die TypKlassen-Funktionen abstützen, und die für alle Typen (die Instanzen der entsprechenden Typklasse sind) funktionieren.
* erhalten Typen durch explizite Instanzbildung (*instance*) oder implizite automatische Instanzbildung

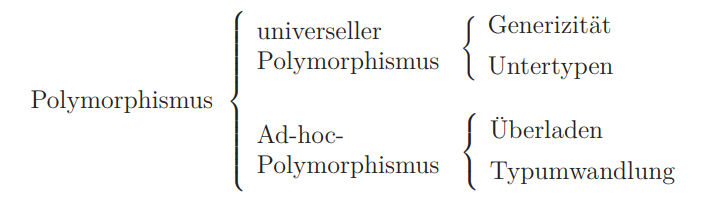
(deriving) als Elemente zugewiesen.

## Polymorphie

Polymorphie = Vieltypigkeit

In Haskell können sowohl Datentypen als auch Funktionen Polymorph deklariert werden, also mit einem Unbestimmten Typen arbeiten, wobei man hier statt dem Typ mit Typvariablen arbeitet.

* Verbesserte Transparenz und Lesbarkeit durch Betonung von Gemeinsamkeiten, nicht von Unterschieden.
* Verlässlichkeit und Wartbarkeit hinsichtlich Fehlersuche, Weiterentwicklung, etc.
* Programmiereffizienz



Nicht alle diese Arten des Polymorphismus gibt es in funktionalen Sprachen.

* + Universell polymorph, echt polymorph, polymorph, Parametrische Polymorphie
  + Ad hoc polymorph, unecht polymorph, Überladen

**Überblick**

Monomorph (Keine Typvariablen, nur konkrete Typen in der Signatur). Rechenvoschriften definiert für genau einen Typ.

fac :: Int -> Int

Parametrisch polymorph (uneingeschränkt polymorph wegen Typvariable, kein Typkontext)

length :: [a] -> Int

Ad hoc polymorph (eingeschränkt polymorph wegen Typvariable und Typkontext )

elem :: Eq a => a -> [a] -> Bool

### Polymorphie auf Funktionen und Datentypen

**Polymorphie auf Datentypen**

Typvariable heißt polymorph, wenn der Wert mit mehreren Datentyp-Deklarationen angegeben werden kann.

Ermöglicht Wiederverwendung durch:

* Datentypkonstruktoren: Dieselbe Struktur für Werte aller Typen.
* Polymorphe Funktionen (Funktionsnamen und -implementierungen) auf diesen Datentypen.

So sind:

Baum Int Int Int,

Baum Int Char String

Baum (a->c) (b->c) (a->b)

Alle Instanzen desselben polymorphen Baum-Typs (a, b, c können beliebige Typen sein):

data Baum a b c = Blatt a b | Wurzel (Baum a b c) c (Baum a b c)

Sprachmittel: Typvariablen, Typklassen

Wiederverwendung: Wiederverwendung von Funktionsname und -implementierung.

**Polymorphie auf Funktionen**

1. **Parametrische Polymorphie**

Es gibt eine einzige Implementierung, die für alle konkreten Typen funktioniert, die für die Typvariablen in der Signatur eingesetzt werden.

Wie Polymorphie auf Datentypen nur mit Funktionen. Angewendet sowohl bei Funktionssignatur als auch bei Pattern- Matching.

Sprachmittel: Typvariablen

1. **Ad-hoc-Polymorphie (Überladen)**

Nur im Zusammenhang mit Typklassen-Funktionen.

Es gibt für jeden Typ, auf dessen Werten die Funktion arbeiten können soll, eine eigene typspezifische Implementierung. Diese typspezifische Implementierung erfolgt durch den Programmierer bei der Instanzbildung eines Typs fur die Typklasse, die diese unecht polymorphe Funktion enthält.

* 1. **Direkt überladene Funktionen**

Funktionen die Typklassenfunktionen erweitern oder überschreiben. Funktion ist Element (Bestandteil) einer Typklasse und wird abhängig vom Kontext / ausführenden Typen ausgewertet.

* 1. **Indirekt überladene Funktionen**

Funktion ist kein Element (Bestandteil) einer Typklasse, stützt sich aber auf eine ab, ohne selbst in einer Typklasse eingeführt zu sein. Funktionen erstellen die nicht Typklassen erweitern oder überschreiben aber ihre Funktionalität nutzen.

sum :: Num a **=>** [a] -> a

Sprachmittel: Typklassen (Haskell spezifisch)

Wiederverwendung: Ad hoc Polymorphie unterstützt Wiederverwendung des Funktionsnamens, nicht jedoch der Funktionsimplementierung (diese wird typspezifisch bei der Typklasseninstanzbildung ausprogrammiert. Man kann nicht eine Funktion für alle Typen die existieren implementieren -> siehe „Grenzen des Überladens“)

### Allgemeinster Typ

Den allgemeinsten Typ jedes Haskell-Ausdrucks kann man auch mit „:t“ bestimmen in der Konsole

length :: [a] -> Int

Ist allgemeinster Typ, der konkreten Typen:

[Int] -> Int

[String] -> Int

[(Float,Float)] -> Int

[(Integer -> Integer)] -> Int

...

Diese Typen sind dadurch Instanzen des Typs ([a] -> Int).

### Grenzen des Überladens

Ist es zum Beispiel möglich jeden Typ zu einer Instanz der Typklasse Eq zu machen?

Dafür müsste man folgende Relation verallgemeinern:

(==) :: Eq a => a -> a -> Bool

In Haskell sind die Typen, auf deren Werten der Gleichheitsrelator (==) definiert ist genau die Elemente (oder Instanzen) der Typklasse Eq. Bei der Eq-Instanzbildung für einen Typ T (gleich ob manuell oder automatisch) wird die typspezifische Bedeutung des Gleichheitsrelators für T-Werte durch explizite Ausprogrammierung von Gleichheits- und Ungleichheitsrelator (==) und (/=) definiert und exakt festgelegt.

z.B.

(==) fac fib ->> False

(==) (\x -> x+x) (\x -> 2\*x) ->> True

(==) (+2) (2+) ->> True

**Antwort - Theorem aus der theoretischen Informatik:**

Gleichheit von Funktionen ist nicht entscheidbar. Es gibt keinen Algorithmus, der für zwei beliebig vorgelegte Funktionen stets nach endlich vielen Schritten entscheidet, ob diese Funktionen gleich sind oder nicht.

(schließt nicht aus, dass für konkret vorgelegte Funktionen oder bestimmte Klassen von Funktionen deren Gleichheit (algorithmisch) entschieden werden kann. -> Nur dass es nicht verallgemeinert werden kann)

* Funktionen lassen sich nicht für jeden Typ angeben, sondern nur für eine Teilmenge aller Typen die Instanzen einer Typklasse sind.

## Rekursions-Arten

In funktionalen Sprachen gibt es keine Anweisungen und deshalb auch keine Schleifen.

Eine Rechenvorschrift heißt rekursiv, wenn sie in ihrem Rumpf (direkt oder indirekt) aufgerufen wird.

Wir unterscheiden zwischen Rekursion auf:

* mikroskopischer Ebene (direkte / unmittelbare Rekursion)

betrachtet einzelne Rechenvorschriften und die syntaktische Gestalt der rekursiven Aufrufe.

1. Repetitive (oder schlichte oder endständige) Rekursion

2. Lineare Rekursion

3. Baumartige (oder kaskadenartige) Rekursion

4. Geschachtelte Rekursion

* makroskopischer Ebene

betrachtet Systeme von Rechenvorschriften und ihre wechselseitigen Aufrufe.

5. Indirekte (verschränkte, wechselweise) Rekursion

**1. Repetitive (schlichte, endständige) Rekursion**

pro Zweig höchstens ein rekursiver Aufruf und diesen stets als äußerste Operation.

Beispiel:

ggt :: Integer -> Integer -> Integer

ggt m n

| n == 0 = m

| m >= n = ggt (m-n) n

| m < n = ggt (n-m) m

**2. Lineare Rekursion**

pro Zweig höchstens ein rekursiver Aufruf, davon mindestens einer nicht als äußerste Operation.

Beispiel:

powerThree :: Integer -> Integer

powerThree n

| n == 0 = 1

| n > 0 = 3 \* powerThree (n-1)

Beachte: In Zweig 2, n > 0, ist “∗” die äußerste Operation, nicht powerThree! -> Weil sie erst danach ausgeführt wird.

**3. Baumartige (kaskadenartige, verzweigte) Rekursion**

pro Zweig können mehrere rekursive Aufrufe nebeneinander vorkommen.

Beispiel:

binom :: Integer -> Integer -> Integer

binom n k

| k == 0 || n == k = 1

| otherwise = binom (n-1) (k-1) + binom (n-1) k

**4. Geschachtelte Rekursion**

rekursive Aufrufe enthalten rekursive Aufrufe als Argumente.

Beispiel:

fun91 :: Integer -> Integer

fun91 n

| n > 100 = n - 10

| n <= 100 = fun91 (fun91 (n+11))

**5. Indirekte (verschränkte, wechselweise) Rekursion**

zwei oder mehr Funktionen rufen sich wechselweise auf.

Beispiel:

isOdd :: Integer -> Bool

isOdd n

| n == 0 = False

| n > 0 = isEven (n-1)

isEven :: Integer -> Bool

isEven n

| n == 0 = True

| n > 0 = isOdd (n-1)

**Effizienz bei Rekursion**

Rekursive Lösungen sind elegant aber nicht immer effizient -> Laufzeit kann schlecht sein.

* Gefahr: (Unnötige) Mehrfachberechnungen
* Besonders anfällig: Baum-/kaskadenartige Rekursion.

Aus Implementierungssicht ist repetitive Rekursion am günstigsten, geschachtelte Rekursion am ungünstigsten.

Deshalb versucht man baumartige und lineare Rekursion auf die repetitive zurückzuführen.

Techniken:

* Akkumulationsparameter – Rechnen auf Parameterposition, ermöglicht die baumartige Rekursion in die günstige repetitive Rekursion überzuführen.
* Dynamische Programmierung
* Memoisation, mit einer Memo-Liste.

Nicht bestimmte Rekursionsmuster an sich sind problematisch, sondern ihr unzweckmäßiger Einsatz, wenn etwa baumartige Rekursion zu (unnötigen) Vielfachberechnungen von Werten führt! Zweckmäßig eingesetzt bietet z.B. baumartige Rekursion viele Vorteile, darunter zur Parallelisierung. Stichwort: Teile und herrsche (oder divide et impera oder divide and conquer)!

**Aufrufgraph**

Knoten sind Funktionen.

Eine einzige Kante falls es einen Aufruf gibt von einer Funktion zur anderen oder die Funktion sich selbst aufruft.

Daraus lässt sich die Art der Rekursivität ablesen.

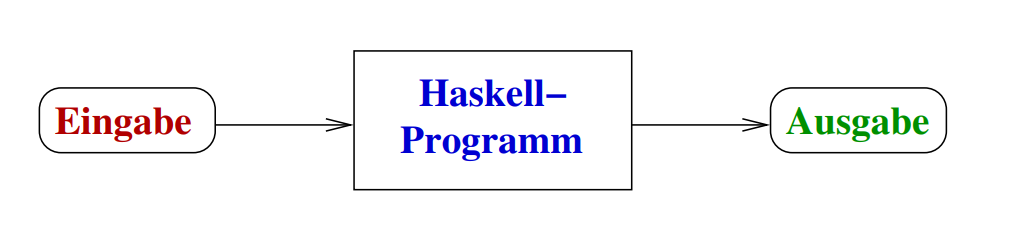
* [A <-> A] bedeutet direkte Rekursivität
* [A <-> B] bedeutet wechselweise Rekursivität
* [A -> B] bedeutet eine direkte hierarchische Abstützung

## I/O

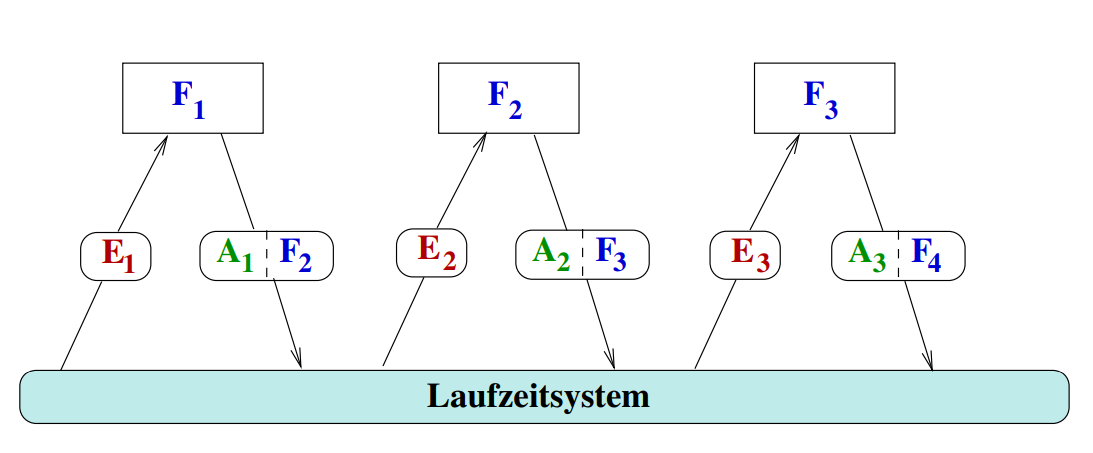
Wir können von der Arbeitweise des Rechners aber nicht des Benutzers abstrahieren. Interaktion ist notwendig.

Durch die Zustandslosigkeit und Seiteneffektfreiheit der funktionalen Programmierung waren unsere Programme sind bislang stapelverarbeitungsorientiert:

* Interaktion zwischen Benutzer und Programm findet nicht statt -> kein I/O
* Eingabedaten müssen zu Programmbeginn vollständig zur Verfügung gestellt werden.
* Einmal gestartet, besteht keine Möglichkeit mehr, mit weiteren Eingaben auf das Verhalten oder Ergebnisse des Programms zu reagieren und es zu beeinflussen.



Wir wollen und interaktionsorientierte Haskell-Programme:



### Probleme in funktionalen Sprachen

**1. Zustandslosigkeit und referentielle Transparenz**

Rein funktionaler Programmierung: Vollkommene Abwesenheit von Seiteneffekten aber Ein-/Ausgabe verlangt Anwesenheit von Seiteneffekten. Ein- und Ausgabe, lesen und schreiben verändern den Zustand der äußeren Welt notwendig und irreversibel.

**2. Eingabe und Ausgabe einer ideellen Schreib- und Lese-Operation**

In funktionaler Programmierung müssen (wie alle Operationen und Funktionen) von funktionalem Typ sein, ein Resultat liefern.

Leseoperationen liefern einen Wert. Den Wert der gelesenen Eingabe.

Schreiboperationen liefern einen Wert.

1. Den geschriebenen Wert
2. einen Wahrheitswert in Abhängigkeit des Erfolgs der Operation
3. irgendeinen Wert (beliebig)
4. festen Wert

**3. Komponierbarkeit**

Ideelle Schreib- und Lese-Funktion lassen sich nicht komponieren.

**3. Zeitliche Anordenbarkeit**

Datenabhängigkeit allein reicht nicht länger aus zur Steuerung der Auswertungsreihenfolge von Ausdrücken!

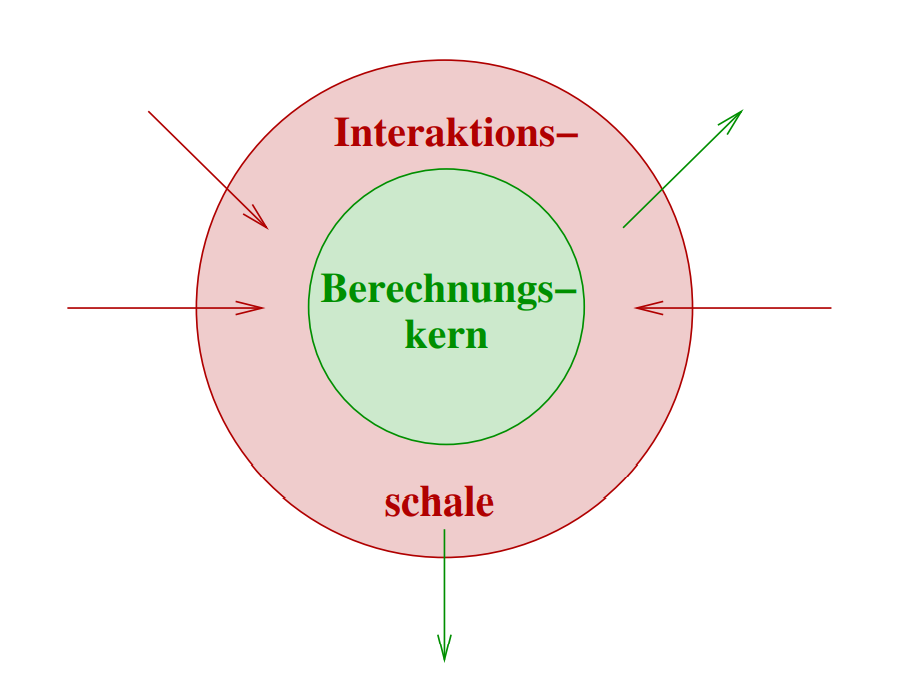
### I/O Lösung in Haskell

Konzeptuell wird in Haskell ein Programm geteilt in

* einen rein funktionalen Berechnungskern
* eine imperativ ähnliche Dialog- und Interaktionsschale. zwischen denen mittels vordefinierter besonderer Ein-/Ausgabefunktionen Daten geschützt ausgetauscht werden können

Haskells Konzept zur Behandlung von Ein-/Ausgabe erlaubt Funktionen

* des Berechnungskerns (rein funktionales Verhalten, keine Seiteneffekte)
* der Dialog- und Interaktionsschale (nicht rein funktionales, sondern seiteneffektbehaftetes Verhalten)



**IO a - Ein vordefinierter polymorpher Datentyp für Ein-/Ausgabe**

In Haskell werden Werte von Ein-/Ausgabeoperationen durch unterschiedliche Typisierung strikt von Werten reiner funktionaler Typen getrennt.

data IO a = ... (Details implementierungsintern versteckt)

Der Datentyp (IO a) erlaubt die Unterscheidung von Typen

* des rein funktionalen Berechnungskerns (Char, Int, Bool, etc.)
* der imperativartigen Dialog- und Interaktionsschale ((IO Char)), (IO Int), (IO Bool), etc.)

IO-Werte bleiben in der Schale u. können nicht den rein funktionalen Kern ‘kontaminieren’.

**return und (<-) - Vermittlungs-Operatoren zwischen Schale und Kern**

Verbindung von funktionalem Kern und E/A-Schale

Von Schale zu Kern: return :: a -> IO a

* ‘*kontaminieren*’ reiner Werte: return erlaubt rein funktionale Werte (engl. pure values) aus dem funktionalen Kern über die Schale als seiteneffektverursachende Werte (engl. impure values) in die äußere Welt zu transferieren.

Return in Haskell hat eine ganz andere Bedeutung als das aus OOP bekannte :

* kann Aktion in auftreten und ausgewertet werden, ohne dass die Auswertung der restlichen Aktionssequenz beendet wird
* kann deshalb auch mehrfach in einer Aktionssequenz auftreten

Von Kern zur Schale: <- :: IO a -> a (Informell)

* ‘*dekontaminieren’* E/A-verschmutzter Werte: I <- erlaubt den ‘reinen’ Anteil (a-Wert) seiteneffektverursachender Werte ((IO a)-Wert) aus der äußeren Welt in den funktionalen Kern zu transferieren.

Bemerkung: return und <- verhalten sich in diesem Sinne invers zueinander, wobei allerdings

* return eine (gewöhnliche) Funktion
* <- einen Wertvereinbarungsoperator (ähnlich ‘:=’ oder ‘=’) aus imperativen, objektorientierten Sprachen bezeichnet aber eigentlich ganz unterschiedlich ist:

‘<-’ bindet einen Wert an einen Namen, bleibt für den gesamten Programmlauf erhalten und ist nicht mehr veränderbar.

‘:=’ leistet eine temporäre Wertzuweisung an eine durch einen Namen bezeichnete Speicherzelle.

Die durch die Zuweisung geschaffene Verbindung zwischen Name (d.h. der mit ihm bezeichneten Speicherzelle) und Wert (d.h. dem Inhalt der Speicherzelle) bleibt so lange erhalten (temporär!), bis sie durch eine erneute Zuweisung an diese Zelle überschrieben und zerstört wird (destruktive Zuweisung!)

**Vordefinierte primitive E/A-Operationen**

als Bausteine, aus denen komplexe(re) Ein-/Ausgabeoperationen gebaut werden können

getChar :: IO Char

getInt :: IO Int

...

putChar :: Char -> IO ()

putInt :: Int -> IO ()

...

**(>>) und (>>=) Kompositionsoperatoren für E/A-Operationen**

Man benötigt eine Komposition von Operatoren damit man wie in einer imperativen Sprache eine Reihenfolge von Aktionen festlegen kann. Dafür eignet sich ein do-Block (als syntactic sugar für >>= ) :

Beispiel:

main = do

foo <- putStrLn "Hello, what's your name?"

name <- getLine

putStrLn ("Hey " ++ name ++ ", you rock!")

(Sonderheit: Die letzte Funktion im do Block muss einen leeren () Wert zurückgeben

denn der Typ einer Aktionssequenz ist durch den Typ der letzten Aktion bestimmt.)

Der Kompositionsoperator (>>=) (bzw die do Notation) erlaubt die präzise Festlegung der zeitlichen Abfolge von E/A-Operationen:

binde-dann-Operator (engl. *bind oder then*)

(>>=) :: IO a -> (a -> IO b) -> IO b

Alternativerweise auch mit einer operationellen Bedeutung:

dann-Operator (engl. *sequence*)

(>>) :: IO a -> IO b -> IO b

Beispiel:

Die Kompositionsoperatoren (>>=) und (>>) erlauben die Bildung (assoziativer) Aktionssequenzen:

akt1 >>= f\_akt2 >> akt3 >> akt4 >>= f\_akt5 >>= return f

### Aktionen

Aktionen sind Ausdrücke vom Typ (IO a).

* sind wertliefernde (‘funktionaler’ Anteil) E/A-Operationen (‘prozeduraler’ Anteil).
* bewirken einen Lese- oder Schreibseiteneffekt (prozedurales Verhalten) und liefern einen a-Wert als Ergebnis (funktionales Verhalten), der eingepackt als (IO a)-Wert zur Verfügung gestellt wird.
* heißen Aktionen (oder Kommandos) (engl. actions oder commands).

Wegen des kombinierten

1. prozeduralen (seiteneffekterzeugende Lese-/Schreiboperation)

2. funktionalen (Wert als Ergebnis liefernden)

Effekts der Auswertung von Aktionen (oder E/A-Ausdrücken), spricht man statt von Auswertung meist von Ausführung von

Aktionen (oder E/A-Ausdrücken).

Informell:

Aktion = (1) E/A-Operation (‘prozedural’) + (2) Wertlieferung (‘funktional’) = wertliefernde E/A-Operation

**Typ aller Leseaktionen**

(IO a) (für ‘lesegeeignete’ Typinstanzen von a)

Der in einen a-Wert transformierte gelesene Wert wird als (formal erforderliches und inhaltlich gewolltes) Ergebnis von Leseoperationen verwendet.

**Typ aller Schreibaktionen**

(IO ()) mit () der einelementige Nulltupeltyp mit gleichbenanntem einzigen Datenwert ().

() als (einziger) Wert des Nulltupeltyps () wird als formal erforderliches Ergebnis von Schreiboperationen verwendet.

**Auswertungsarten**

1. rein funktionaler Ausdruck (engl. pure expression): Der Wert des Ausdrucks wird geliefert (‘funktionaler’ Effekt), sonst (passiert) nichts.
2. Aktion bzw E/A-Ausdruck (engl. impure expression):

1. Eine E/A- Operation wird ausgeführt (Lese-/Schreibseiteneffekt wird generiert, ‘prozeduraler’ Effekt).

2. ein a-Wert (eingepackt in den Datenwertkonstruktor IO) wird als Wert des E/A-Ausdrucks geliefert (‘funktionaler’ Effekt).

**Kompositionsarten**

1. Funktionskomposition (.)

Schreiben mit Zeilenvorschub (vordefinierte Sequenz):

putStrLn :: String -> IO ()

putStrLn = putStr . (++ "\n")

1. IO-Komposition (>>=)

Lesen einer Zeile und anschließendes Schreiben der

gelesenen Zeile (selbstdefinierte Sequenz):

echo :: IO ()

echo = getLine >>= (\zeile -> putLine zeile)

### Iterativ-artige E/A-Programme

**do-Notation – vereinfachte Komposition**

Die do-Notation als Ersatz für die IO-Kompositionsoperatoren (>>=) und (>>) zur imperativ ähnlicheren Bildung von Ein-/Ausgabesequenzen:

Zwei Beispiele:

do zeile <- getLine statt getLine >>= (\zeile -> putStrLn zeile)

putStrLn zeile

do putStr "fun" statt putStr "fun" >> putStr "\n"

putStr "\n" putStr "fun" >>= (\\_ -> putStr "\n")

Ein do-Ausdruck entspricht semantisch einer Sequenz von E/A-Operationen und kann (deshalb) auf eine beliebige Anzahl von Aktionen als Argumente angewendet werden.

Die Bedeutung des do-Ausdrucks:

do w1 <- akt1

w2 <- akt2

...

wn <- aktn

return ( f w1 w2 ... wn )

ist definiert durch den (>>=)-Ausdruck:

akt1 >>= \p1 ->

akt2 >>= \p2 ->

...

aktn >>= \pn ->

return ( f p1 p2 ... pn )

**Typ des do-Ausdrucks**

Der Typ eines do-Ausdrucks ist durch den Typ seiner letzten Aktion bestimmt.

Aktionen liefern stets ein Ergebnis. Bleibt es unverwendet (entspricht Aktionskomposition mit (>>) statt mit (>>=)), kann die Nichtverwendung syntaktisch dadurch ausgedrückt werden, dass ein Aktionsergebnis nicht an einen Wertnamen wi, sondern an ‘gebunden’ wird, quasi ‘unbenannt’ gebunden wird:

do w1 <- akt1

\_ <- akt2

\_ <- akt3

w4 <- akt4

...

wn <- aktn

return (f w1 w4 ... wn)

**while Funktion**

Ersetzt if else Verzweigung

while :: IO Bool -> IO () -> IO ()

while bedingung aktion

= do b <- bedingung

if b

then

do aktion

while bedingung aktion -- Rekursion!

else

return ()

**Einstiegspunkt**

Einstiegspunkt für die Auswertung (übersetzter) interaktiver Haskell-Programme ist (per Konvention) eine eindeutig bestimmte

* Definition mit Namen main vom Typ (IO a)

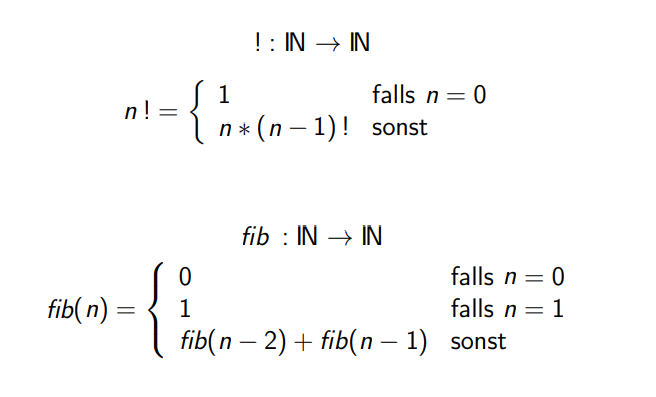
Intuitiv: ‘Haskell-Programm = E/A-Aktion’.

## Fehlerbehandlung

Bisher bei Fehlern Berechnung mit dem Aufruf error "Ungültige Eingabe" unterbrochen.

### Partialität von Funktionen

Natürliche Funktionen die auf natürlichen Zahlen totaldefiniert sind:



Ihre Implementierungen sind hingegen häufig nur **partiell definiert**.

So terminieren folgende Implementierungen nur für nichtnegative Argumente und sind für negative Argumente nicht definiert.

Explizite, transparente Sichtbarmachung der Partialität ist wichtig.

fac :: Int -> Int

fac n

| n == 0 = 1

| n >= 1 = n \* fac (n - 1)

| otherwise = **error "undefiniert"**

fib :: Int -> Int

fib n

| n == 0 = 1

| n == 1 = 1

| n >= 2 = fib (n-2) + fib (n-1)

| otherwise = **error "undefiniert"**

Wenn Funktionen die Partialität verstecken und undefinierte Eingaberäume nicht behandeln, dann sind sie „intransparent“ und „verschleiernd“.

### Fehlerbehandlung

**Panikmodus (engl. panic mode)**

polymorphe Funktion error :: String -> a meldet Fehler und stoppt Auswertung.

Beispiel:

fac :: Integer -> Integer

fac n

| n == 0 = 1

| n > 0 = n \* fac (n-1)

| otherwise = error "Ungültige Eingabe."

Positiv:

* Generell, einfach zu implementieren, schnell

Negativ: Radikalität

* Die Berechnung stoppt unwiderruflich.
* Jegliche Information über den Programmlauf ist verloren, auch sinnvolle.
* Für sicherheitskritische Systeme können die Folgen eines unbedingten Programmabbruchs fatal sein.

**Auffangwerte (engl. default values)**

Auffangwerte zur Weiterrechnung im Fehlerfall.

1. **Funktionsspezifisch**

im Fehlerfall wird ein funktionsspezifischer Wert als Resultat geliefert.

Beispiel:

fac :: Integer -> Integer

fac n

| n == 0 = 1

| n > 0 = n \* fac (n-1)

| otherwise = -1

Im Beispiel von fac gilt:

* Negative Werte treten nie als reguläres Resultat einer Berechnung auf.
* Der funktionsspezifische Auffangwert −1 erlaubt deshalb, negative Eingaben als fehlerhaft zu erkennen und zu melden, ohne den Programmlauf unwiderruflich abzubrechen.

Positiv

* Panikmodus vermieden, Programmlauf nicht abgebrochen.

Negativ

* Kann das Eintreten der Fehlersituation verschleiern und intransparent machen, wenn der Auffangwert auch als Resultat einer regulären Berechnung auftreten kann.
* unintuitiv
* Oft fehlt ein funktionsspezifischer Auffangwert gänzlich

1. **Aufrufspezifisch**

Im Fehlerfall wird ein aufrufspezifischer Auffangwert als Resultat geliefert.

Die Fehlersituation ist für den Programmierer transparent.

Allerdings: Nicht immer taugt das Argument selbst als Auffangwert. In solchen Fällen ist folgender allgemeinere Ansatz nötig.

Beispiel:

fac :: Integer -> Integer

fac n

| n == 0 = 1

| n > 0 = n \* fac (n-1)

| otherwise = n

Man kann auch Fehler an eine dritte Funktion delegieren.

Positiv:

* Panikmodus vermieden, Programmlauf nicht abgebrochen.
* Generalität, stets anwendbar.
* Flexibilität, aufrufspezifische Auffangwerte ermöglichen variierende Fehlerwerte und Fehlerbehandlung.

Negativ:

* Transparente Fehlerbehandlung ist nicht gewährleistet, wenn aufrufspezifische Auffangwerte auch reguläres Resultat einer Berechnung sein können – In diesen Fällen Gefahr ausbleibender Fehlerwahrnehmung Folgen durch Vortäuschen eines regulären und korrekten Berechnungsablaufs und eines regulären und korrekten Ergebnisses!

#### Zusammengefasst

-- 1) error -> exception/"panic mode"

f 0 = 0

f \_ = error "wrong argument"

-- 2) special values

f :: Int -> Int

f 0 = 0

f \_ = -1 --or a value that tells more about cause of error like n

-- 3.1) special (wrapper) type that brings type safety and makes possibility of failure lear

f :: Int -> Maybe Int

f 0 = Just 0

f \_ = Nothing

-- 3.2) basically same as above, but allows(/requires) passing on an error message type ErrMsg = String

f :: Int -> Either Int ErrMsg

f 0 = Left 0

f \_ = Right "wrong argument"

### Fehlertypen, Fehlerwerte, Fehlerfunktionen

**(Fehler-) Datentyp Maybe a**

Anzeigbarkeit von Fehlern wird erreicht durch Übergang von Typ a zum (Fehler-) Datentyp Maybe a:

data Maybe a = Just a | Nothing deriving (Eq, Ord, Read, Show)

umfasst die Werte des Typs a in der Form Just a mit dem Zusatzwert Nothing als explizitem Fehlerwert.

Positiv:

* Fehler können erkannt, angezeigt, weitergereicht und schließlich gefangen und (im Sinn von Auffangwertvariante 2) behandelt werden.
* Systementwicklung ist ohne explizite Fehlerbehandlung möglich (z.B. mit nichtfehlerbehandelnden Funktionen wie div).
* Fehlerbehandlung kann nach Abschluss durch Ergänzung der fehlerbehandelnden Funktionsvarianten zusammen mit den Funktionen map\_Maybe und maybe durchgereicht werden

Negativ:

* Geänderte Funktionalität: Maybe b statt b.

Pragmatische Zusatzvorteile:

**Die Funktion map\_Maybe**

Beachte: map\_Maybe ist verschieden von der im prelude definierten namensähnlichen Funktion mapMaybe mit Signatur:

mapMaybe :: (a -> Maybe b) -> [a] -> [b]

Die Signatur von map\_Maybe:

map\_Maybe :: (a -> b) -> Maybe a -> Maybe b

map\_Maybe f Nothing = Nothing --Durchreichen von Fehlern

map\_Maybe f (Just u) = Just (f u) --Rechnen mit f im Normalfall

**Die Funktion maybe**

maybe :: b -> (a -> b) -> Maybe a -> b

maybe x f Nothing = x -- (Aufrufspez. Fehlerwert)

maybe x f (Just u) = f u -- (Rechnen mit f im Normalfall)

**Zusammenfassung von Fehlerfunktionen**

Im Zusammenspiel erlauben map Maybe und maybe Fehlerwerte

* Weiterzureichen

Map\_Maybe: map\_Maybe f Nothing = Nothing

der Fehlerwert Nothing wird von map Maybe durchgereicht.

* zu fangen und (im Sinn von Auffangwertvariante 2) zu behandeln

maybe x f Nothing = x

der aufrufspezifische Auffangwert x wird als Resultat geliefert

Im Zusammenspiel erlauben map Maybe und maybe Fehlerwerte

## Module

Zerlegung von Programmen in überschaubare, (oft) getrennt übersetzbare Programmeinheiten als wichtige programmiersprachliche Unterstützung der Programmierung im Großen.

Zwei wichtige Eigenschaften zur Charakterisierung guter Modularisierung:

* Kohäsion (modullokal, intramodular) beschäftigt sich mit dem inneren Zusammenhang von Modulen, mit Art und Typ der in einem Modul zusammengefassten Funktionen.
* Koppelung (modulübergreifend, intermodular) beschäftigt sich mit dem äußeren Zusammenhang von Modulen, dem Import-/Export- und Datenaustauschverhalten.

**Kennzeichen gelungener Modularisierung**

Starke funktionale und Datenkohäsion

enger inhaltlicher Zusammenhang der Definitionen eines Moduls.

Schwache funktionale und lose Datenkoppelung

wenige Abhängigkeiten zwischen verschiedenen Modulen, insbesondere keine direkten oder indirekten zirkulären Abhängigkeiten.

**Aufbau von Haskell Modulen**

Moduldateien werden eingeleitet von der Zeile:

EXPORT:

module M where -- ermöglicht Export von M

module M1 (D\_1 (..), D\_2, T\_1, f\_2, f\_5) where -- selektiver Export von M

IMPORT:

module M where

import M1 -- nicht selektiver Import

module M where

import M1 (D\_1 (..), D\_2, T\_1, C\_1 (..), C\_2, f\_5) -- selektiver Import

module M where

import M1 hiding (D\_1, T\_2, f\_1) -- selektiver Import

gefolgt von Deklarationen/Definitionen von:

1. Typen (algebraische Typen, Neue Typen, Typsynonyme)

2. Typklassen

3. Funktionen

Selektiver Import und Export bedeutet, dass der Programmierer festlegen kann, welche Namen (Datentypen, Konstruktoren, Funktionsnamen, Wertenamen, etc.) von anderen Modulen importiert werden können.

**Reexport**

Der Verzicht auf automatischen Reexport von Namen bedeutet, dass Namen, die ein Modul aus anderen Modulen importiert hat, nicht automatisch für den Import durch ein anderes Modul zur Verfügung stehen. Sie müssen explizit zum Reexport zugelassen werden.

**Namenskonflikte**

Namenskonflikte können durch qualifizierten Import aufgelöst werden:

import qualified M1

Verwendung: M1.f zur Bezeichnung der aus M1 importierten Funktion f

f zur Bezeichnung der im importierenden Modul lokal definierten Funktion f

Umbenennen importierter Module und Bezeichner durch Einführen lokaler Namen im importierenden Modul – für Modulnamen:

import qualified M1 as MyLocalNameForM1

MyLocalNameForM1 wird im importierenden Modul anstelle von M1 verwendet.

Alternativerweise, wenn selektiv:

import M1 (f1,f2) renaming (f1 to fac, f2 to fib)

### Abstrakte Datentypen mit Modulen

**Konkrete Datentypen (KDT) (in Haskell: Algebr. Datentypen)**

* werden durch die exakte Angabe und Darstellung ihrer Werte spezifiziert, aus denen sie bestehen.
* auf ihnen gegebene Funktionen/Operationen werden zum Definitionszeitpunkt nicht angegeben und bleiben offen.

**Abstrakte Datentypen (ADT)**

* werden durch ihr Verhalten spezifiziert, d.h. durch die auf ihren Werten definierten Funktionen/Operationen und deren Zusammenspiel.
* die tatsächliche Darstellung der Werte des Datentyps wird zum Definitionszeitpunkt nicht angegeben u. bleibt offen.

**Implementierung des ADT in Haskell**

Implementierungstechnischer Schlüssel: Haskells Modulkonzept, speziell der selektive Export, bei dem Konstruktoren algebraischer Datentypen verborgen bleiben wodurch das mit ADT-Definitionen verfolgte Ziel:

* Kapselung von Daten, Realisierung des Geheimnisprinzips auf Datenebene (engl. information hiding) erreicht werden kann.

## Entwicklungs-Stil: Reflektive Programmierung

Einflechtung reflektiver Feedback Schleifen im Programmentwicklungsprozess:

Wasserfall-Entwicklungsmodell mit 4 Stufen wobei man ständig zurückhüpfen kann.

(a) Problem verstehen

(b) Problemlösung konzipieren

(c) Problemlösung implementieren

(d) Implementierte Problemlösung testen und evaluieren

Für jede dieser Phasen gibt es typische Fragen. Von ihrer Beantwortung hängt es ab, ob mit der nächsten Phase fortgefahren wird oder in die oder eine frühere Phase zurückgesprungen wird. Dieses kontinuierliche Nachdenken und Hinterfragen des aktuellen Ergebnisses des bisherigen Entwicklungsprozesses ist namensgebend für dieses Vorgehen: Reflektive Programmierung.