

# Funktionale Programmierung

LVA 185.A03, VU 2.0, ECTS 3.0  
WS 2020/2021

Vortrag V  
Orientierung, Einordnung  
21.11.2020

Jens Knoop



Technische Universität Wien  
Information Systems Engineering  
Compilers and Languages



Vortrag V

Teil V

Kap. 12

Kap. 13

Kap. 14

Umgekehrte  
Klassen-  
zim-  
mer IV

Hinweis

Aufgabe

# Vortrag V

## Orientierung, Einordnung

*...zum selbstgeleiteten, eigenständigen Weiterlernen.*

### Teil V: Fundierung funktionaler Programmierung

- Kapitel 12:  $\lambda$ -Kalkül
  - ↪ Berechenbar(keitstheorie), Churchsche These, **einfachste funktionale (Programmier-) Sprache**,...
- Kapitel 13: Auswertungsordnungen
  - ↪ normal, applikativ, faul, Church/Rosser-Resultate,...
- Kapitel 14: Typprüfung, Typinferenz
  - ↪ monomorph, polymorph, mit Typklassen, Unifikation,...

*“...much of our attention is focused on **functional programming**, which is the most successful programming paradigm founded on a rigorous mathematical discipline. **Its foundation, the lambda calculus, has an elegant computational theory and is arguably the smallest universal programming language.** As such, the lambda calculus is also crucial to understand the properties of language paradigms other [than] functional programming...”*

Exzerpt von der Startseite der  
‘Programming Languages and Systems (PLS)’  
Forschungsgruppe an der University of New South Wales,  
Sydney, geleitet von Manuel Chakravarty und Gabriele Keller.  
( <http://www.cse.unsw.edu.au/~pls/PLS/PLS.html> )

# Teil V

## Fundierung funktionaler Programmierung

# Kapitel 12

## $\lambda$ -Kalkül

Vortrag V

Teil V

**Kap. 12**

12.1

12.2

12.3

12.4

12.5

12.6

Kap. 13

Kap. 14

Umgekehrte  
Klassen-  
zim-  
mer IV

Hinweis

Aufgabe

# Kapitel 12.1

## Motivation

Vortrag V

Teil V

Kap. 12

**12.1**

12.2

12.3

12.4

12.5

12.6

Kap. 13

Kap. 14

Umgekehrte  
Klassen-  
zim-  
mer IV

Hinweis

Aufgabe

# Der $\lambda$ -Kalkül: Eines verschiedener

...universeller formaler Berechenbarkeitsmodelle:

- Turing-Maschinen
- Markov-Algorithmen
- Allgemein rekursive Funktionen
- ...

...fundamental in der Berechenbarkeitstheorie:

- Was heißt berechenbar?
- Was ist berechenbar? Welche Probleme sind berechenb.?
- Wie aufwändig ist etwas zu berechnen? Was ist effizient berechenbar? In Theorie? In Praxis?
- Gibt es Grenzen der Berechenbarkeit, des Berechenbaren?
- ...

...darüberhinaus: Formale Grundlage und Basis

- funktionaler Programmierung
- funktionaler Programmiersprachen

# Berechenbar, B.keit. Was kann das sein?

...eine umfassende, aber vollkommen informelle Vorstellung:

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

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

Ist mit diesem Begriff etwas für die Beantwortung der Fragen der Berechenbarkeitstheorie gewonnen?

- ↪ Auf den ersten Blick nichts: Die Bedeutungen von
- 'etwas' und 'irgendwie machbar'
- und damit von intuitiv berechenbar sind vollkommen vage und nicht greifbar, nichts als ein nebulöses Bauchgefühl!

Aber...



## ...es wird ein 1) Ziel & ein 2) Weg aufgezeigt!

1) Das Ziel: Das 'Wesen' von 'etwas', von 'irgendwie machbar' und damit von intuitiv berechenbar

- formal fassbar u. (damit) präzise behandelbar zu machen.

2) Der Weg zum Ziel: Ersinne/erfinde effektive mechanische Methoden  $M$ , die für

- gültige Argumentwerte in endlich vielen Schritten den Funktionswert konstruieren
- nicht gültige Argumentwerte mit einem besonderen Fehlerwert oder nie abbrechen.

⇒ Die 'Belohnung': Jede solche Methode  $M$  definiert ein

- formales Berechenbarkeitsmodell mit einem formalen Berechenbarkeitsbegriff: Berechenbar mit  $M$ ,  $M$ -berechenbar

... $M$ ,  $M$ -berechenbar können beide mathematisch rigoros untersucht werden!

# Damit eröffnet sich ein Weg zur Überprüfung

...ob das Ziel erreicht ist!

## Behaupte:

Theorem (bitte nach mir als Erfinder v.  $M$  benennen)

'Etwas' ist intuitiv berechenbar gdw es ist  $M$ -berechenbar!

Solch kühne Behauptung gehört bewiesen:

$\Leftarrow$ : Trivial! Nichts zu tun!

$\Rightarrow$ : Unmöglich! Was heißen 'etwas' und intuitiv berechenbar?

Was bleibt? Enttäuschung, Demut, Bescheidenheit:

These (bitte trotzdem nach mir benennen!)

'Etwas' ist intuitiv berechenbar gdw es ist  $M$ -berechenbar.

# Was bleibt noch? Vergleichbarkeit!

...rigorose Vergleichbarkeit von Methoden  $M$ ,  $M'$  bzgl. ihrer  
– (Berechnungs-) Stärke, Ausdruckskraft.

Dadurch möglich: **Falsifizierbarkeit!**

**Falsifizierbarkeit** der  $M$ -These für jedes  $M$ :

Die  $M$ -These, dass intuitiv berechenbar und  $M$ -berechenbar ident seien, ist widerlegt, wenn eine berechnungsstärkere, ausdruckskräftigere Methode  $M'$  als  $M$  gefunden wird, wenn also etwas  $M'$ -berechenbar ist, aber nicht  $M$ -berechenbar.

Einfacher sogar: ...wenn ein konkretes Problem, eine konkrete Aufgabe oder Fragestellung gefunden wird, die *ad hoc*, auf der Stelle als berechenbar eingesehen werden kann, aber nicht  $M$ -berechenbar ist.

...hingegen ist **Verifizierbarkeit** der  $M$ -These für jedes  $M$  **unmöglich!**

# Bsp. wichtiger formaler Berechnungsmethoden

...als Kern zugehöriger formaler Berechnungsmodelle und ihre zeitliche Einordnung:

- Allgemein rekursive Funktionen (Herbrand 1931, Gödel 1934, Kleene 1936) (s. Anh. B.3, B.4)
- $\mu$ -rekursive Funktionen (Kleene 1936) (s. Anh. B.4)
- Turing-Maschinen (Turing 1936) (s. Anh. B.1)
- Endliche kombinatorische Prozesse (Post 1936)
- Markov-Algorithmen (Markov 1951) (s. Anh. B.2)
- Registermaschinen (Random Access Machines (RAMs)) (Shepherdson, Sturgis 1963)
- ...
- $\lambda$ -definierbare Funktionen des  $\lambda$ -Kalküls (Church 1936)

# Zentrales Resultat der Berechenbarkeitstheorie

## Theorem 12.1.1 (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!

## Korollar 12.1.2 (Universalität des $\lambda$ -Kalküls)

Alles, was in einem der vorher genannten Modelle berechenbar ist, ist im  $\lambda$ -Kalkül berechenbar (und umgekehrt!).

# Holzschnittartige Charakterisierung, Einteilung

...dieser formalen Berechnungsmodelle und ihrer Methoden:

- Maschinenbasierte/-orientierte Konkretisierungen von 'berechenbar':
  - Turing-Maschinen
  - Registermaschinen
  - ...
- Programmierbasierte/-orientierte Konkretisierungen von 'berechenbar':
  - Markov-Algorithmen
  - Theorie rekursiver Funktionen
    - allgemein rekursiv
    - $\mu$ -rekursiv
  - ...
  - $\lambda$ -definierbare Funktionen des  $\lambda$ -Kalküls.

...jede dieser Methoden  $M$  induziert eine These, die  $M$ -These.

# Zu den häufigst angeführten zählt...

...die Formulierung als sog.:

## Churchsche These (' $\lambda$ -Kalkülthese')

'Etwas' ist intuitiv berechenbar gdw es ist im  $\lambda$ -Kalkül berechenbar.

Zur Churchschen oder Church/Turingschen These siehe z.B.:

B. Jack Copeland. [The Church-Turing Thesis](#). The Stanford Encyclopedia of Philosophy, 2002.

<http://plato.stanford.edu/entries/church-turing>

# Die konzeptuelle Verschiedenheit der Modelle

...legt nahe, [Theorem 12.1.1](#) als [starken Hinweis](#) (keinesfalls als Beweis!) darauf zu interpretieren und zu verstehen, dass alle diese [Modelle](#) den Begriff

- [‘etwas’](#) ist [intuitiv berechenbar](#) wahrscheinlich [‘gut’](#)

charakterisieren; [gut](#) im Sinne von [vollständig](#) und [umfassend](#).

[Jedoch](#): [Theorem 12.1.1](#) schließt nicht aus, dass vielleicht noch heute eine

- [mächtigere \(Berechnungs-\) Methode](#)

gefunden wird, die [‘etwas’](#) und [intuitiv berechenbar](#) [umfassender](#), [vollständiger](#) und damit [besser](#) charakterisierte; oder auch nur

- [ein \(!\) Problem](#)

dass offensichtlich berechenbar ist, aber nicht im  $\lambda$ -Kalkül.



# Präzedenzfall

...das Berechenbarkeitsmodell

- primitiv rekursiver Funktionen

galt bis 1928 als adäquate Charakterisierung für 'etwas' und intuitiv berechenbar. Jedoch:

- Primitiv-rekursiv-berechenbar ist echt schwächer als im  $\lambda$ -Kalkül berechenbar, Turing-berechenbar, Markov-berechenbar, ,...

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.

(Zur Definition des Schemas primitiv rekursiver Funktionen siehe z.B.: Wolfram-Manfred Lippe. Funktionale und Applikative Programmierung. eXamen.press, 2009, Kapitel 2.1.2.)

# Die Ackermann-Funktion

...hier in einer Formulierung von Rózsa Péter aus dem Jahr 1935 in Haskell-Notation:

```
ack :: (Integer,Integer) -> Integer
ack (m,n)
  | m == 0           = n+1
  | (m > 0) && (n == 0) = ack (m-1,1)
  | (m > 0) && (n /= 0) = ack (m-1,ack(m,n-1))
```

...‘berühmtberüchtigtes’ Beispiel einer offensichtlich

- effektiv berechenbaren, damit auch intuitiv berechenbaren, aber nicht primitiv-rekursiv-berechenbaren Funktion.

# Andere Frage: ‘Etwas’ “gut genug” erfasst?

...durch die Festlegung “‘etwas’ ist intuitiv berechenbar, wenn es eine ‘irgendwie machbare’ effektive mechanische Methode gibt, die für

- gültige Argumentwerte in endlich vielen Schritten den Funktionswert konstruiert
- ...”

wird eine

- funktionsorientierte Vorstellung von ‘etwas’

induziert, die Berechnungsmodellen wie dem  $\lambda$ -Kalkül offensichtlich zugrundeliegt und implizit die Problemtypen festlegen könnte, die überhaupt als

- Berechnungsproblem

aufgefasst werden (können).

# Konkret: Ist Interaktion von 'etwas' umfasst?

...sind Leistungen, Aktivitäten, Tätigkeiten, die

- Betriebssysteme
- Eingebettete Systeme (Steuerung, Überwachung)
- Internet (dynamisch hinzutretende, ausscheidende Dienste)
- Cyberphysikalische Systeme (Roboter, Drohnen,...)
- ...

erbringen,

- Fahrzeuge autonom ihren Weg im realen Straßenverkehr zu vorgegebenen Zielen finden lassen
- ...

durch eine funktionsorientierte Vorstellung von 'etwas' umfasst, d.h. vorstellbar, darstellbar mit einmaliger Eingabedatenbereitstellung ohne irgendeine weitere Interaktion?

# Naheliegende Fragen im Fall von

...Betriebs- und Steuerungssystemen:

- Ist die Berechnung, Verarbeitung endlich? Terminiert sie?
- Welche Funktion wird berechnet?

...dem Internet:

- Können Systeme mit flexibel hinzutretenden und ebenso wieder wegfallenden Komponenten als statisch angesehen werden? Wenn ja, in welchem Sinn?
- Welche Funktion wird berechnet?

...autonomen Fahrzeugen:

- Wie sehen Ein- und Ausgabe aus?
- Welche Funktion wird berechnet?

Ist Interaktion für Aufgaben dieser Art nicht unverzichtbar?

# Anders ausgedrückt

...ändert **Hinzunahme von Interaktion** das Verständnis von **Berechnung**, **berechenbar**, **Berechenbarkeit** möglicherweise ähnlich grundlegend wie die Findung der **Ackermann-Funktion?**

Angestoßen wurden Fragen und Forschung hierzu besonders durch:

- ▶ Peter Wegner. **Why Interaction is More Powerful Than Algorithms**. Communications of the ACM 40(5):81-91, 1997.

Darunter liegen erneut die Fragen:

- Was ist **Berechnung**, was ist das **Wesen** von **Berechnung?**
- Was heißt **berechenbar?**
- Was ist **berechenbar?**

# Sind Antworten wie z.B. in:

- ▶ Martin Davis. *What is a Computation?* Kapitel in Lynn A. Steen (Hrsg.), *Mathematics Today – Twelve Informal Essays*. Springer-V., 241-268, 1978.

...ausreichend? Oder müssen sie angepasst u. weiterentwickelt werden angesichts der Realität **massiv parallelen, verteilten, interaktiven, asynchronen, analogen, Echtzeit-, hybriden Rechnens, des Rechnens mit neuronalen Netzen, chemischen Reaktionen, biologischen Systemen (Bakterien), des Quanten-, Nano-, DNS-Rechnens, des Rechnens mit Molekülen, Enzymen,...**?

- ▶ Luca Cardelli. *Programming with Chemical Reactions*. VCLA-Kolloquiumsvortrag an der TU Wien, 22.11.2018.

“Chemical reactions have been widely used to describe natural phenomena, but increasingly we are capable to use them to prescribe physical interaction, e.g. in DNA computing. Thus, chemical reaction networks can be used as programs that can be physically realized to produce and control molecular arrangements. Because of their relative simplicity and familiarity, and more subtly because of their computational power, they are quickly becoming a paradigmatic “programming language” for bioengineering. We discuss what can be programmed with chemical reactions, and how these programs can be physically realized.”

# Die zentrale Frage von Peter Wegner

...auf den Punkt gebracht :

Gilt die Church/Turing-These im **schwachen Sinn**:

- Was immer durch eine Funktion im math. Sinn intuitiv berechenbar ist, d.h. wann immer es eine effektive mechanische Methode für ihre Berechnung gibt, kann von einer Turing-Maschine, im  $\lambda$ -Kalkül, etc. berechnet werden.

...oder im **starken Sinn**:

- Was immer eine 'Berechnungsmaschine' ('Computer'), ggf. mithilfe von Interaktion, berechnen kann, kann von einer Turing-Maschine, im  $\lambda$ -Kalkül berechnet werden, d.h. wann immer – auch über Funktionen im math. Sinn hinaus – eine Aufgabe als Berechnung ausgedrückt und verstanden werden kann, kann sie von einer Turing-Maschine, im  $\lambda$ -Kalkül, etc. berechnet werden.



# Schwache vs. starke Church/Turing-These (1)

...eine 'für' und 'wider' untersuchte Frage:

- ▶ Michael Prasse, Peter Rittgen. **Why Church's Thesis Still Holds. Some Notes on Peter Wegner's Tracts on Interaction and Computability.** The Computer Journal 41(6):357-362, 1998.
- Peter Wegner, Eugene Eberbach. **New Models of Computation.** The Computer Journal 47(1):4-9, 2004.
- Paul Cockshott, Greg Michaelson. **Are There New Models of Computation? Reply to Wegner and Eberbach.** The Computer Journal 50(2):232-247, 2007.
- ▶ Dina Q. Goldin, Peter Wegner. **The Interactive Nature of Computing: Refuting the Strong Church-Rosser Thesis.** Minds and Machines 18(1):17-38, 2008.

## Schwache vs. starke Church/Turing-These (2)

- ▶ Peter Wegner, Dina Q. Goldin. *The Church-Turing Thesis: Breaking the Myth*. In Proceedings of the 1st Conference on Computability in Europe – New Computational Paradigms (CiE 2005), Springer-V., LNCS 3526, 152-168, 2005.
- ▶ Martin Davis. *The Church-Turing Thesis: Consensus and Opposition*. In Proceedings of the 2nd Conference on Computability in Europe – Logical Approaches to Computational Barriers (CiE 2006), Springer-V., LNCS 3988, 125-132, 2006.
- ▶ ...

Wer richtig erkennen will, muss zuvor  
in richtiger Weise gezweifelt haben.

Aristoteles (384 - 322 v.Chr.)  
griech. Philosoph

# Interaktion und Berechenbarkeit

...hat **Interaktion** das Potential zu einer neuen **Ackermannfunktion**-vergleichbaren **Weltsichtänderung**, was **berechenbar heißt**, was **berechenbar ist**? Eine offene Frage...

Ich weiß, das klingt alles sehr kompliziert.

Fred Sinowatz (1929-2008)

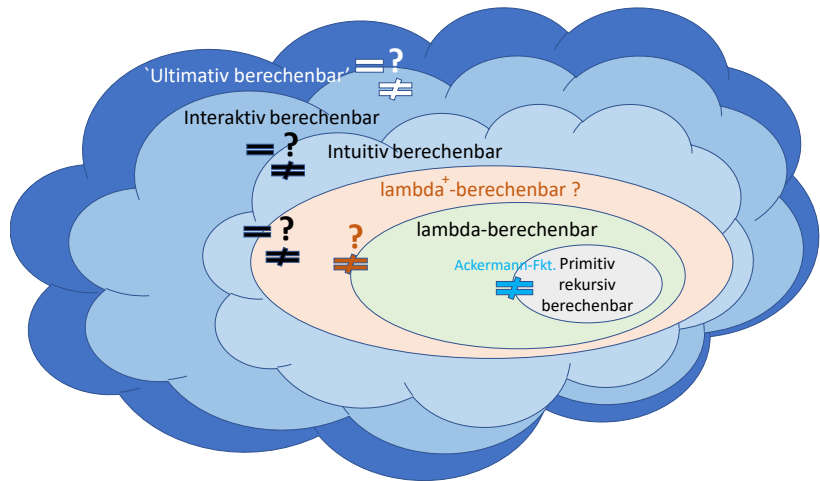
österr. Politiker und Staatsmann, Bundeskanzler

Es ist nicht leicht zu begreifen,  
dass man nicht begreift.

Marie von Ebner-Eschenbach (1830-1916)

österr. Schriftstellerin

# Die Welt d. Berechenbaren: Wie sieht sie aus?



# Leseempfehlungen

...als Klassiker:

- ▶ John McCarthy. [A Basis for a Mathematical Theory of Computation](#). In *Computer Programming and Formal Systems*, Paul Braffort, David Hirschberg (Hrsg.), North-Holland, 33-70, 1963.

...für neue Sichten:

- ▶ S. Barry Cooper, Benedikt Löwe, Andrea Sorbi (Hrsg.). [New Computational Paradigms: Changing Conceptions of What is Computable](#). Springer-V., 2008.

...als gute und knappe Einführung:

- ▶ Ian Horswill. [What is Computation?](#) Crossroads, the ACM Magazine for Students 18(3):8-14, 2012.

...weitere Literaturhinweise in [Anhang B](#).

# Forschung, Diskurs

...gehen weiter; für einen **kompakten Einstieg** in das Themenfeld zusammen mit Verweisen auf wichtige Arbeiten siehe:

- ▶ B. Jack Copeland, Eli Dresner, Diane Proudfoot, Oron Shagrir. **Viewpoint: Time to Reinspect the Foundations? Questioning if Computer Science is Outgrowing its Traditional Foundations.**  
Communications of the ACM 59(11):34-36, 2016.
- ▶ B. Jack Copeland, Oron Shagrir. **The Church-Turing Thesis: Logical Limit or Breachable Barrier?**  
Communications of the ACM 62(1):66-74, 2019.

Nur Beharrung führt zum Ziel.

Friedrich von Schiller (1759-1805)  
dt. Schriftsteller

# Zu schwer?

Man kann viel, wenn man sich nur recht viel zutraut.

Wilhelm von Humboldt (1767-1835)  
dt. Philosoph und Politiker

Vortrag V

Teil V

Kap. 12

12.1

12.2

12.3

12.4

12.5

12.6

Kap. 13

Kap. 14

Umgekehrte  
Klassen-  
zim-  
mer IV

Hinweis

Aufgabe

# Zurück zum $\lambda$ -Kalkül

## Der $\lambda$ -Kalkül

- geht zurück auf [Alonzo Church](#) (1936).
- ist erdacht als [formales Berechnungsmodell](#) (neben anderen), um Fragen der Natur von Berechnung und dessen, was berechnet werden kann, zu fassen u. zu untersuchen.
- formalisiert einen [Berechnungsbegriff](#) über Paaren, Listen, Bäumen, auch potentiell unendlichen, über Funktionen höherer Ordnung, etc.
- ist in diesem Sinne durch [größere Praxisnähe](#) als (einige) andere formale Berechnungsmodelle ausgezeichnet.
- wurde mit der Erfindung von [Rechenanlagen](#) und [Programmiersprachen](#) zur/zum
  - Grundlage aller [funktionalen Programmiersprachen](#).
  - Bindeglied [funktionaler Hochsprachen](#) und [maschinennaher Implementierungen](#).



# Eigenschaften des $\lambda$ -Kalküls

Der  $\lambda$ -Kalkül zeichnet sich aus durch

- Einfachheit  
...wenige syntaktische Konstrukte, einfache Semantik.
- Ausdruckskraft  
...Turing-mächtig, alle 'intuitiv berechenbaren' Funktionen sind im  $\lambda$ -Kalkül berechenbar.

In diesem Sinn ist der  $\lambda$ -Kalkül ein

- universelles Berechnungsmodell

und Grundlage aller

- funktionalen Programmiersprachen, quasi die 'Assembler-Sprache' funktionaler Programmierung.

# Wichtige Anwendungsfelder des $\lambda$ -Kalküls

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 $\lambda$ -Kalkül, angewandte $\lambda$ -Kalküle

## Reiner $\lambda$ -Kalkül

- Reduziert auf das ‘absolut Notwendige’, angemessen und bedeutsam besonders für grundlegende Untersuchungen zu Fragen der Berechenbarkeit, [Berechenbarkeitstheorie](#).

## Angewandte $\lambda$ -Kalküle

- Syntaktisch angereicherte Varianten des reinen  $\lambda$ -Kalküls, [praxis-](#) und [programmiersprachennäher](#).

[Extrem angereicherte angewandte  \$\lambda\$ -Kalküle](#) nennen wir

- [Funktionale Programmiersprachen](#).

# Kapitel 12.2

## Syntax des reinen $\lambda$ -Kalküls

Vortrag V

Teil V

Kap. 12

12.1

**12.2**

12.3

12.4

12.5

12.6

Kap. 13

Kap. 14

Umgekehrte  
Klassen-  
zim-  
mer IV

Hinweis

Aufgabe

# Syntax von $\lambda$ -Ausdrücken (1)

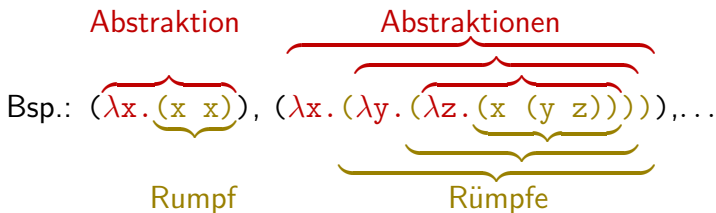
Die Menge  $E$  (von engl. Expression) der wohlgeformten Ausdrücke des reinen  $\lambda$ -Kalküls über einer Menge  $N$  von Namen, kurz  $\lambda$ -Ausdrücke, ist induktiv wie folgt definiert:

1. **Namen:** Jeder Name aus  $N$  ist in  $E$ .

Bsp.:  $a, b, c, \dots, x, y, z, \dots$

2. **Abstraktionen:** Ist  $x$  aus  $N$ ,  $e$  aus  $E$ , so ist  $(\lambda x. e)$  in  $E$ .

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



# Syntax von $\lambda$ -Ausdrücken (2)

3. **Applikationen:** 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**.

Bsp.:  $((\underbrace{\lambda x. (x x)}_{\text{Rator}}) \underbrace{y}_{\text{Rand}}), \dots$

# Syntax von $\lambda$ -Ausdrücken in BNF-Notation

...Ausdrucksyntax kompakt in Backus-Naur-Form (BNF):

$e ::= x$	(Namen)
$::= \lambda x.e$	((Funktions-) Abstraktion)
$::= e e$	((Funktions-) Applikation)
$::= (e)$	(Klammerung)

**Informell:** Der  $\lambda$ -Kalkül bietet **Namen**, (**anonyme**) **Funktionen** (auch höherer Ordnung!) und **Funktionsanwendungen**, durch **Klammerung** auch in geschachtelter Form.

**Anmerkung:** Statt **Name** aus **N** sagt man oft auch **Variable** aus **V** und identifiziert **N** und **V** miteinander.

# Vereinbarungen, Konventionen

Überflüssige Klammern können weggelassen werden. Es gilt:

- ▶ Rechtsassoziativität für  $\lambda$ -Sequenzen in Abstraktionen.

Beispiele:

- $\lambda x. \lambda y. \lambda z. (x (y z))$  steht kurz für  $(\lambda x. (\lambda y. (\lambda z. (x (y z))))))$
- $\lambda x. e$  steht kurz für  $(\lambda x. e)$

- ▶ Linksassoziativität für Applikationssequenzen.

Beispiele:

- $e_1 e_2 e_3 \dots e_n$  steht kurz für  $(\dots ((e_1 e_2) e_3) \dots e_n)$
- $e_1 e_2$  steht kurz für  $(e_1 e_2)$

Der Rumpf einer  $\lambda$ -Abstraktion ist der längstmögliche dem Punkt folgende  $\lambda$ -Ausdruck.

Beispiel:  $\lambda x. e f$  steht kurz für  $\lambda x. (e f)$ , nicht  $(\lambda x. e) f$



# Bindungsbereich, Gültigkeitsbereich v. Namen

...in  $\lambda$ -Ausdrücken.

- Der **Bindungsbereich** der gebundenen Variablen einer  $\lambda$ -Abstraktion ist der **Rumpf** der  $\lambda$ -Abstraktion.

$$\underbrace{(\lambda f. \lambda g. \lambda h. f (g h))}_{\text{Bindungsbereich von } f} \underbrace{\lambda g. ((\lambda g. g) g)}_{\text{Bindungsbereich von } g}$$

Bindungsbereich von  $g$

- Der **Gültigkeitsbereich** der gebundenen Variablen einer  $\lambda$ -Abstraktion ist ihr **Bindungsbereich** abzüglich aller echt inneren **Bindungsbereiche** mit gleichnamiger gebundener Variable.

$$\underbrace{(\lambda f. \lambda g. \lambda h. f (g h))}_{\text{Gültigkeitsbereich von } f} \underbrace{\lambda g. ((\lambda g. g) g)}_{\text{Gültigkeitsbereich von } g}$$

Gültigkeitsbereich von  $g$

# Freie, gebundene Variablen (1)

...in  $\lambda$ -Ausdrücken. Sei  $a$  aus  $E$ :

Freie Variablen von  $a$ :

$$\text{frei}(x) = \{x\} \quad \text{wenn } a \equiv x \text{ aus } N$$

$$\text{frei}(\lambda x.e) = \text{frei}(e) \setminus \{x\} \quad \text{wenn } a \equiv \lambda x.e$$

$$\text{frei}(f e) = \text{frei}(f) \cup \text{frei}(e) \quad \text{wenn } a \equiv f e$$

Gebundene Variablen von  $a$ :

$$\text{gebunden}(x) = \emptyset \quad \text{wenn } a \equiv x \text{ aus } N$$

$$\text{gebunden}(\lambda x.e) = \text{gebunden}(e) \cup \{x\} \quad \text{wenn } a \equiv \lambda x.e$$

$$\begin{aligned} \text{gebunden}(f e) &= \text{gebunden}(f) \\ &\quad \cup \text{gebunden}(e) \quad \text{wenn } a \equiv f e \end{aligned}$$

Anm.:  $\equiv$  steht für lexikalisch gleich.

## Freie, gebundene Variablen (2)

Beispiel: Betrachte den  $\lambda$ -Ausdruck  $((\lambda x. (x y)) x)$ .

Gesamtausdruck:

- $x$  kommt in  $((\lambda x. (x y)) x)$  frei und gebunden vor.
- $y$  kommt in  $((\lambda x. (x y)) x)$  frei vor, aber nicht gebunden.

Teilausdrücke:

- $x$  kommt in  $(\lambda x. (x y))$  gebunden vor, aber nicht frei.
- $x$  kommt in  $(x y)$  und  $x$  frei vor, aber nicht gebunden.
- $y$  kommt in  $(\lambda x. (x y))$ ,  $(x y)$  und  $y$  frei vor, aber nicht gebunden.

Beachte: 'Gebunden' und 'frei' sind nicht Negationen voneinander (anderenfalls gälte z.B. 'x kommt gebunden in y vor' oder 'y kommt frei in  $\lambda x. x$  vor', was beides nicht der Fall ist).

# Freie, gebundene Variablenvorkommen

...in  $\lambda$ -Ausdrücken:

- **Definierende** Vorkommen: Jedes Variablenvorkommen unmittelbar nach einem  $\lambda$ .
- **Angewandte** Vorkommen: Jedes nicht definierende Variablenvorkommen.
- **Gebunden an**: Relation zwischen Variablenvorkommen und definierenden Variablenvorkommen. Jedes Variablenvorkommen (gleich ob angewandt oder definierend) ist an höchstens ein definierendes Variablenvorkommen gebunden; definierende Vorkommen sind an ihr Vorkommen selbst gebunden.
- **Freies Variablenvorkommen**: Angewandtes Vorkommen, das an kein definierendes Vorkommen gebunden ist.
- **Gebundenes Variablenvorkommen**: Vorkommen (gleich ob angewandt oder definierend), das an ein definierendes Vorkommen gebunden ist.

# Kapitel 12.3

## Semantik des reinen $\lambda$ -Kalküls

Vortrag V

Teil V

Kap. 12

12.1

12.2

**12.3**

12.3.1

12.3.2

12.3.3

12.3.4

12.3.5

12.3.6

12.4

12.5

12.6

Kap. 13

Kap. 14

Umgekehr

Klassen-  
zim-  
mer IV

Hinweis

Aufgabe

...für die Definition der Semantik von  $\lambda$ -Ausdrücken sind:

- Syntaktische Substitution
- Konversionsregeln/Reduktionsregeln
- Reduktionsfolgen/Reduktionsstrategien
- Normalformen (Existenz/Eindeutigkeit)

# Kapitel 12.3.1

## Syntaktische Substitution

Vortrag V

Teil V

Kap. 12

12.1

12.2

12.3

**12.3.1**

12.3.2

12.3.3

12.3.4

12.3.5

12.3.6

12.4

12.5

12.6

Kap. 13

Kap. 14

Umgekehrte  
Klassen-  
zim-  
mer IV

Hinweis

Aufgabe

# Syntaktische Substitution

...eine dreistellige **Abbildung**

$$\cdot[\cdot/\cdot] : E \rightarrow E \rightarrow V \rightarrow E$$

zur **bindungsfehlerfreien** Ersetzung der freien Vorkommen einer Variablen  $x$  durch einen Ausdruck  $e$  in einem Ausdruck  $e'$ .

**Vereinfachend, informell:** 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.

**Beachte:** Die vereinfachende informelle Beschreibung nimmt keinen Bedacht auf mögliche **Bindungsfehler**. Freiheit von Bindungsfehlern stellt die formale Definition **syntaktischer Substitution** sicher.



# Syntaktische Substitution

## Definition 12.3.1.1 (Syntaktische Substitution)

Die **syntaktische Substitution** ist die 3-stellige Abbildung

$\cdot[\cdot/\cdot] : E \rightarrow E \rightarrow V \rightarrow E$  definiert bei Anwendung auf

...**Namensterme** durch:

$$y[e/x] =_{df} \begin{cases} y & \text{falls } y \text{ aus } N \text{ mit } y \neq x \\ e & \text{falls } y \text{ aus } N \text{ mit } y = x \end{cases}$$

...**applikative Terme** durch:

$$(f g)[e/x] =_{df} (f[e/x]) (g[e/x])$$

...**Abstraktionsterme** durch:

$$(\lambda y.f)[e/x] =_{df} \begin{cases} \lambda y.f & \text{falls } y = x \\ \lambda y.(f[e/x]) & \text{falls } y \neq x \wedge y \notin \text{frei}(e) \\ \lambda z.((f[z/y])[e/x]) & \text{falls } y \neq x \wedge y \in \text{frei}(e), \\ & \text{wobei } z \text{ frisch aus } N: z \notin \text{frei}(e) \cup \text{frei}(f) \\ & \text{(Vermeidung von Bindungsfehlern!)} \end{cases}$$

# Beispiel: Einfache Anwendungen

...syntaktischer Substitution:

$$- ((x \ y) (y \ z)) [(a \ b)/y] = ((x \ (a \ b)) ((a \ b) \ z))$$

$$- \lambda x. (x \ y) [(a \ b)/y] = \lambda x. (x \ (a \ b))$$

$$- \lambda x. (x \ y) [(a \ b)/x] = \lambda x. (x \ y)$$

# Beispiel: Bindungsfehler

Ein **Bindungsfehler** entsteht bei naiver Anwendung **syntaktischer Substitution**, wenn ein Ausdruck mit einer freien Variable in den Gültigkeitsbereich einer gebundenen Variable gleichen Namens eingesetzt wird:

- $\lambda x. (x y) [(x b)/y] \xrightarrow{\text{naiv}} \lambda x. (x (x b))$ : **Bindungsfehler!**  
...naiv ohne Umbenennung angewendet ist  $x$  eingefangen!

Korrekt mit **Umbenennung** angewendet gibt es **keinen Bindungsfehler**:

$$\begin{aligned} - \lambda x. (x y) [(x b)/y] &= \lambda z. ((x y)[z/x]) [(x b)/y] \\ &\quad x \text{ frei in } (x b) \text{ Umbenennung von } x \text{ in } z \\ &= \lambda z. (z y) [(x b)/y] \\ &\quad \text{Umbenannt} \\ &= \lambda z. (z (\underbrace{x}_b) b) \end{aligned}$$

**Kein Bindungsfehler:**  $x$  in  $(x b)$  bleibt frei!

# Kapitel 12.3.2

## Konversionsregeln

Vortrag V

Teil V

Kap. 12

12.1

12.2

12.3

12.3.1

**12.3.2**

12.3.3

12.3.4

12.3.5

12.3.6

12.4

12.5

12.6

Kap. 13

Kap. 14

Umgekehrte  
Klassen-  
zimmer  
IV

Hinweis

Aufgabe

# $\lambda$ -Konversionsregeln, $\lambda$ -Konversionen

...die  $\lambda$ -Konversionsregeln führen abgestützt auf syntaktische Substitution zu einer operationellen Semantik für  $\lambda$ -Ausdrücke in Form maximaler Ausdrucksvereinfachung:

## Definition 12.3.2.1 ( $\lambda$ -Konversionsregeln)

1.  $\alpha$ -Konversion (Umbenennung von Parametern)

$$\lambda x. e \longleftrightarrow \lambda y. e [y/x], \text{ wobei } y \notin \text{frei}(e)$$

2.  $\beta$ -Konversion (Funktionsanwendung)

$$(\lambda x. f) e \longleftrightarrow f [e/x]$$

3.  $\eta$ -Konversion (Elimination redundanter Funktion)

$$\lambda x. (e x) \longleftrightarrow e, \text{ wobei } x \notin \text{frei}(e)$$

# Zur Anwendung der Konversionsregeln

$\alpha$ -Konversion zur

- konsistenten Umbenennung von Parametern von  $\lambda$ -Abstraktionen (zur Vermeidung von Bindungsfehlern (s.u.)).

$\beta$ -Konversion zur

- Anwendung einer  $\lambda$ -Abstraktion auf ein Argument.

$\eta$ -Konversion zur

- Elimination unnötiger  $\lambda$ -Abstraktionen.

Erinnerung: Naiv ohne  $\alpha$ -Konversion angewendet, kann die Substitution der  $\beta$ -Konversion Bindungsfehler verursachen, wie im nachstehenden Beispiel illustriert:

Bsp.:  $(\lambda x. (\lambda y. x y)) (y z) \rightarrow (\lambda y. x y)[(y z)/x] \rightarrow (\lambda y. (y z) y)$   
(ohne  $\alpha$ -Konversion ist  $y$  eingefangen: Bindungsfehler!)

...korrekt angewendet: Keine Bindungsfehler dank  $\alpha$ -Konversion!

# Konversionsregeln, Reduktionsregeln

...Sprechweisen im Zusammenhang mit Konversionsregeln:

- Von links nach rechts angewendet: Reduktion.
- Von rechts nach links angewendet: Abstraktion.

Genauer:

- Von links nach rechts gerichtete Anwendungen der  $\beta$ - und  $\eta$ -Konversion heißen  $\beta$ -Reduktion und  $\eta$ -Reduktion.
- Von rechts nach links gerichtete Anwendungen der  $\beta$ -Konversion heißen  $\beta$ -Abstraktion.

Entsprechend spricht man von  $\beta$ - und  $\eta$ -Reduktionsregeln, um auszudrücken, dass die entsprechenden Konversionsregeln nur von links nach rechts angewendet werden.

# Kapitel 12.3.3

## Reduktionsfolgen

Vortrag V

Teil V

Kap. 12

12.1

12.2

12.3

12.3.1

12.3.2

**12.3.3**

12.3.4

12.3.5

12.3.6

12.4

12.5

12.6

Kap. 13

Kap. 14

Umgekehrte  
Klassen-  
zim-  
mer IV

Hinweis

Aufgabe



# Reduktionsfolgen, -ordnungen, Normalform

Eine **Reduktionsfolge** für einen  $\lambda$ -Ausdruck

- ist eine endliche oder nicht endliche Folge von  $\beta$ -,  $\eta$ -Reduktionen und  $\alpha$ -Konversionen.
- heißt **maximal**, wenn höchstens noch  $\alpha$ -Konversionen anwendbar sind.

(Grund-) **Reduktionsordnungen, -strategien** sind

- Normale Reduktion(sordnung) (äußerst)
- Applikative Reduktion(sordnung) (innerst)

Praktisch relevante **Reduktionsordnungen, -strategien** sind

- Linksnormale Reduktion(sordnung) (linkest-äußerst)
- Linksapplikative Reduktion(sordnung) (linkest-innerst)

Ein  $\lambda$ -Ausdruck ist in **Normalform**, wenn er

- durch  $\beta$ -,  $\eta$ -Reduktionen nicht weiter reduzierbar ist.

# Beispiele zu Reduktionsfolgen, -ordnungen (1)

## Beispiel 1: Applikative Ordnung

$$\underbrace{((\lambda z. \lambda y. (z y))}_{\text{Rator}} \underbrace{(\lambda x. x)}_{\text{Rand}}) (\lambda s. (s s))$$

$$(\beta\text{-Reduktion}) \longrightarrow \underbrace{(\lambda y. ((\lambda x. x) y))}_{\text{Rator}} \underbrace{(\lambda s. (s s))}_{\text{Rand}}$$

$$(\beta\text{-Reduktion}) \longrightarrow \underbrace{(\lambda x. x)}_{\text{Rator}} \underbrace{(\lambda s. (s s))}_{\text{Rand}}$$

$$(\beta\text{-Reduktion}) \longrightarrow \lambda s. (s s)$$

...fertig, Normalform erreicht: Keine  $\beta$ -,  $\eta$ -Reduktion mehr anwendbar.

# Beispiele zu Reduktionsfolgen, -ordnungen (2)

## Beispiel 2: Applikative Ordnung

$$((\lambda x. \lambda y. x y) ((\underbrace{(\lambda x. \lambda y. x y)}_{\text{Rator}}) \underbrace{a}_{\text{Rand}}) b)) c$$

$$(\beta\text{-Reduktion}) \longrightarrow (\lambda x. \lambda y. x y) (\underbrace{(\lambda y. a y)}_{\text{Rator}} \underbrace{b}_{\text{Rand}}) c$$

$$(\beta\text{-Reduktion}) \longrightarrow (\underbrace{(\lambda x. \lambda y. x y)}_{\text{Rator}}) \underbrace{(a b)}_{\text{Rand}} c$$

$$(\beta\text{-Reduktion}) \longrightarrow \underbrace{(\lambda y. (a b) y)}_{\text{Rator}} \underbrace{c}_{\text{Rand}}$$

$$(\beta\text{-Reduktion}) \longrightarrow (a b) c$$

...fertig, Normalform erreicht: Keine  $\beta$ -,  $\eta$ -Reduktion mehr anwendbar.

# Beispiele zu Reduktionsfolgen, -ordnungen (3)

## Beispiel 2': Normale Ordnung

$$\begin{aligned} & \underbrace{((\lambda x. \lambda y. x y))}_{\text{Rator}} \underbrace{(((\lambda x. \lambda y. x y) a) b)}_{\text{Rand}} c \\ (\beta\text{-Reduktion}) & \longrightarrow \underbrace{(\lambda y. (((\lambda x. \lambda y. x y) a) b) y)}_{\text{Rator}} \underbrace{c}_{\text{Rand}} \\ (\beta\text{-Reduktion}) & \longrightarrow \underbrace{(((\lambda x. \lambda y. x y) a)}_{\text{Rator}} \underbrace{b)}_{\text{Rand}} c \\ (\beta\text{-Reduktion}) & \longrightarrow \underbrace{((\lambda y. a y))}_{\text{Rator}} \underbrace{b)}_{\text{Rand}} c \\ (\beta\text{-Reduktion}) & \longrightarrow (a b) c \end{aligned}$$

...fertig, Normalform erreicht: Keine  $\beta$ -,  $\eta$ -Reduktion mehr anwendbar.

# Kapitel 12.3.4

## Normalformen

Vortrag V

Teil V

Kap. 12

12.1

12.2

12.3

12.3.1

12.3.2

12.3.3

**12.3.4**

12.3.5

12.3.6

12.4

12.5

12.6

Kap. 13

Kap. 14

Umgekehr

Klassen-  
zim-  
mer IV

Hinweis

Aufgabe

# Normalformen

...existieren nicht notwendig; nicht jeder  $\lambda$ -Ausdruck

- besitzt eine Normalform, ist in Normalform konvertierbar.

Beispiel: Der Ausdruck

$$\underbrace{\lambda x.(x x)}_{\text{Rator}} \underbrace{\lambda x.(x x)}_{\text{Rand}} \longrightarrow \lambda x.(x x) \lambda x.(x x) \longrightarrow \dots$$

...reproduziert sich durch fortgesetzte  $\beta$ -Reduktionen  
endlos: Eine Normalform existiert nicht!

# Reduktionsfolgen

...terminieren nicht notwendig mit einem

- $\lambda$ -Ausdruck in Normalform, auch wenn eine existiert.

Beispiel:

$$- \underbrace{(\lambda x.y)}_{\text{Rator}} \underbrace{(\lambda x.(x x) \lambda x.(x x))}_{\text{Rand}} \longrightarrow y$$

Normale Reduktion terminiert in einem Schritt mit Normalform!

$$- (\lambda x.y) \underbrace{(\lambda x.(x x))}_{\text{Rator}} \underbrace{\lambda x.(x x)}_{\text{Rand}} \longrightarrow (\lambda x.y) \underbrace{(\lambda x.(x x))}_{\text{Rator}} \underbrace{\lambda x.(x x)}_{\text{Rand}}$$

$\longrightarrow \dots$

Applikative Reduktion terminiert nicht, obwohl Normalform existiert!

# Hauptresultate: Die Church/Rosser-Theoreme

Seien  $e_1, e_2$  zwei  $\lambda$ -Ausdrücke.

## Theorem 12.3.4.1 (Konfluenz-, Diamant-, Rauteneig.)

Wenn  $e_1, e_2$  ineinander konvertierbar sind, d.h.  $e_1 \longleftrightarrow e_2$ , dann gibt es einen gemeinsamen  $\lambda$ -Ausdruck  $e$ , zu dem  $e_1, e_2$  reduziert werden können, d.h.  $e_1 \longrightarrow^* e$  und  $e_2 \longrightarrow^* e$ .

**Informell:** Wenn eine **Normalform** existiert, dann ist sie (bis auf  $\alpha$ -Konversion) **eindeutig** bestimmt!

## Theorem 12.3.4.2 (Standardisierung)

Wenn  $e_1$  zu  $e_2$  mit einer endlichen Reduktionsfolge reduzierbar ist, d.h.  $e_1 \longrightarrow^* e_2$ , und  $e_2$  in Normalform ist, dann führt auch die normale Reduktionsfolge von  $e_1$  nach  $e_2$ .

**Informell:** Normale Reduktion **terminiert am häufigsten**, so oft wie überhaupt nur möglich!



# Folgerungen

...aus den **Church/Rosser-Theoremen** (Alonzo Church, John Barkley Rosser (1936)):

- **Theorem 12.3.4.1** garantiert, dass die **Normalform** eines  $\lambda$ -Ausdrucks (bis auf  $\alpha$ -Konversionen) **eindeutig** bestimmt ist, wenn sie existiert;  $\lambda$ -Ausdrücke in Normalform lassen sich (abgesehen von  $\alpha$ -Konversionen) nicht mehr weiter reduzieren, vereinfachen.
- **Theorem 12.3.4.2** garantiert, dass die **normale Reduktionsordnung** mit der Normalform **terminiert**, wenn es **irgendeine** Reduktionsfolge mit dieser Eigenschaft gibt, d.h. die **normale** Reduktionsordnung terminiert mindestens so häufig wie jede andere Reduktionsstrategie, mit hin **am häufigsten**.

*Omnia viae Romam ducunt.  
Alle Wege führen zum Ergebnis  
(wenn sie denn zum Ergebnis führen).*

lat., sprichwörtl., abgewandelt

# Kapitel 12.3.5

## Semantik von $\lambda$ -Ausdrücken

Vortrag V

Teil V

Kap. 12

12.1

12.2

12.3

12.3.1

12.3.2

12.3.3

12.3.4

**12.3.5**

12.3.6

12.4

12.5

12.6

Kap. 13

Kap. 14

Umgekehr

Klassen-  
zim-  
mer IV

Hinweis

Aufgabe

# Semantik von $\lambda$ -Ausdrücken

...die Church/Rosser-Theoreme und ihre Garantien legen nahe, die Semantik (oder Bedeutung) der Ausdrücke des reinen  $\lambda$ -Kalküls in folgender Weise festzulegen:

## Definition 12.3.5.1 (Semantik von $\lambda$ -Ausdrücken)

Sei  $e$  ein  $\lambda$ -Ausdruck. Die Semantik von  $e$  ist

- seine (bis auf  $\alpha$ -Konversionen) eindeutig bestimmte Normalform, wenn sie existiert; die Normalform ist die Bedeutung und der Wert von  $e$ .
- undefiniert, wenn die Normalform nicht existiert.

# Determiniertheit, Turingmächtigkeit

## Lemma 12.3.5.2 (Determiniertheit)

Wenn ein  $\lambda$ -Ausdruck in einen  $\lambda$ -Ausdruck in Normalform konvertierbar ist, dann führt jede terminierende Reduktionsfolge des  $\lambda$ -Ausdrucks (bis auf  $\alpha$ -Konversion) zu dieser Normalform, d.h. das Resultat jeder terminierenden Reduktionsfolge ist (bis auf  $\alpha$ -Konversion) determiniert.

## Theorem 12.3.5.3 (Turingmächtigkeit)

Eine Funktion ist im  $\lambda$ -Kalkül genau dann berechenbar, wenn sie Turing-berechenbar, Markov-berechenbar, etc., ist, d.h. im  $\lambda$ -Kalkül sind alle Funktionen berechenbar, die Turing-berechenbar, Markov-berechenbar, etc., sind und umgekehrt.

# Kapitel 12.3.6

## Rekursion vs. Y-Kombinator

Vortrag V

Teil V

Kap. 12

12.1

12.2

12.3

12.3.1

12.3.2

12.3.3

12.3.4

12.3.5

**12.3.6**

12.4

12.5

12.6

Kap. 13

Kap. 14

Umgekehr

Klassen-  
zim-  
mer IV

Hinweis

Aufgabe

# Rekursion

...ist im reinen  $\lambda$ -Kalkül nicht vorgesehen!

Betrachte die **argumentfrei** definierte **rekursive Haskell-Rechen-**vorschrift **fac**:

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

Im  $\lambda$ -Kalkül (wie in **Haskell**) stellt sich folgendes Problem:

- $\lambda$ -Abstraktionen sind **anonym** und können deshalb **nicht (rekursiv) aufgerufen** werden:

```
 $\lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n * \dots ??? \dots$ 
```

...es gibt keinen Namen, den wir für '???' einsetzen können.

**Rekursive Aufrufe** wie im Rumpf von **fac** lassen sich deshalb im reinen  $\lambda$ -Kalkül **nicht** ausdrücken.

# Abhilfe: Kombinatoren, der Y-Kombinator

... $\lambda$ -Terme ohne freie Variablen heißen **Kombinatoren**.

Der **Y-Kombinator**, ein spezieller **Kombinator**:

$$Y = \lambda f. (\lambda x. (f (x x)) \lambda x. (f (x x)))$$

...ein **Kombinator** mit **Selbstanwendung**:

$$Y = \lambda f. (\underbrace{\lambda x. (f (x x))}_{\text{Rator}} \underbrace{\lambda x. (f (x x))}_{\text{Rand}})$$

...**Rator** ident mit **Rand**: **Selbstanwendung!**

und der **Fähigkeit**, sich zu **reproduzieren**, zu **kopieren!**

# Schlüsselfähigkeit des Y-Kombinators

...Selbstreproduktion plus Argumentkopie!

Für  $e$   $\lambda$ -Ausdruck ist  $(Y e)$  zu  $(e (Y e))$  konvertierbar:

$$\begin{aligned} Y e &\longleftrightarrow \underbrace{(\lambda f. (\lambda x. (f (x x)) \lambda x. (f (x x))))}_{= Y} e \\ &\longrightarrow \underbrace{\lambda x. (e (x x)) \lambda x. (e (x x))}_{= Y e} \\ &\longrightarrow e \underbrace{(\lambda x. (e (x x)) \lambda x. (e (x x)))}_{= e (Y e)} \\ &\longleftrightarrow \underbrace{e}_{\text{Kopie}} \underbrace{(Y e)}_{\text{Selbstreproduktion}} \end{aligned}$$

...Selbstreproduktion plus Kopie des Arguments  $e$ !



# Der Y-Kombinator

...ermöglicht es, **Rekursion** durch

– Kopieren

zu ersetzen und zu realisieren.

**Idee:** Überführe eine **rekursive** Darstellung von **f** in eine **nicht-rekursive** Darstellung, die den **Y-Kombinator** verwendet:

$$\begin{aligned} f &= \dots f \dots && \text{(Rekursive Darstellung von } f \text{)} \\ \rightsquigarrow f &= \lambda f. (\dots f \dots) f && \text{(}\lambda\text{-Abstraktion)} \\ \rightsquigarrow f &= \underbrace{Y \lambda f. (\dots f \dots)}_{\text{'Y e'}} && \text{(Nichtrekursive Darstellung von } f \text{)} \end{aligned}$$

# Übungsaufgabe 12.3.6.1

1. **Analogie:** Vergleiche den Effekt des **Y-Kombinators** mit der **Kopierregelsemantik** prozeduraler Programmiersprachen.
2. **Anwendung des Y-Kombinators:** Gegeben ist die rekursionsfreie Darstellung von **fac** mit **Y-Kombinator**:

$$\text{fac} = Y \lambda f. (\lambda n. \text{if } n == 0 \text{ then } 1 \text{ else } n * f (n - 1))$$

- 2.1 **Rechne nach**, dass sich der Term **(fac 1)** auf den Normalformterm **1** reduzieren lässt:

$$\text{fac } 1 \longrightarrow \dots \longrightarrow 1$$

- 2.2 **Überprüfe** dabei, dass durch den **Y-Kombinator** Rekursion durch wiederholtes Kopieren ersetzt ist.

# Kapitel 12.4

## Angewandte $\lambda$ -Kalküle

Vortrag V

Teil V

Kap. 12

12.1

12.2

12.3

**12.4**

12.5

12.6

Kap. 13

Kap. 14

Umgekehrte  
Klassen-  
zim-  
mer IV

Hinweis

Aufgabe

# Angewandte $\lambda$ -Kalküle

...sind syntaktisch angereicherte Varianten des reinen  $\lambda$ -Kalküls.

In **Ausdrücken** angewandter  $\lambda$ -Kalküle können

- **Konstanten**, **Funktionsnamen**, 'übliche' **Operatoren** ähnlich wie Namen auftreten und an die Seite von  $\lambda$ -Abstraktionen treten:

42, 3.14, true, false, +, \*, -, fac, binom, ...

- neue Ausdrücke als Abkürzungen eingeführt und verwendet werden:

**cond** e e<sub>1</sub> e<sub>2</sub> , **if** e **then** e<sub>1</sub> **else** e<sub>2</sub>, ...

- **Typen** auftreten, Ausdrücke **getypt** sein:

42 : **IN**, 3.14 : **IR**, true : **IBool**, ...

- ...

# Wohlgeformte Ausdrücke

...angewandter  $\lambda$ -Kalküle können also auch

– Applikative Terme wie

$2+3$ ,  $\text{fac } 3$ ,  $\text{fib } (2+3)$ ,  $\text{binom } x \ y$ ,  $((\text{binom } x) \ y)$ , ...

– Abstraktionsterme wie

$\lambda x. (x + x)$ ,  $\lambda x. \lambda y. \lambda z. (x * (y - z))$ ,  
 $(\lambda x. \text{if odd } x \text{ then } x * 2 \text{ else } x \text{ div } 2)$ , ...

sein, für deren Auswertung zusätzliche **Reduktionsregeln** eingeführt werden, sog.:

–  $\delta$ -Reduktionen

...für die **Auswertung**, **Reduktion** arithmetischer Ausdrücke, bedingter Ausdrücke, Operationen auf Listen, etc.

# Beispiel

... $\delta$ -Reduktionsfolge: Unecht 'applikative' Ordnung  
(unecht, da ohne Verzahnung von  $\beta$ -,  $\eta$ - und  $\delta$ -Reduktionen)

$(\lambda x. \lambda. x * y) ((\lambda x. \lambda y. x + y) 9 5) 3$

( $\beta$ -Reduktion, li)  $\longrightarrow (\lambda x. \lambda y. x * y) ((\lambda y. 9 + y) 5) 3$

( $\beta$ -Reduktion, li)  $\longrightarrow (\lambda x. \lambda y. x * y) (9 + 5) 3$

( $\beta$ -Reduktion, li)  $\longrightarrow (\lambda y. (9 + 5) * y) 3$

( $\beta$ -Reduktion, li)  $\longrightarrow (9 + 5) * 3$

...keine  $\beta$ -,  $\eta$ -Reduktion mehr anwendbar; weiter mit  $\delta$ -Reduktionen:

...  $(9 + 5) * 3$

( $\delta$ -Reduktion, li)  $\longrightarrow 14 * 3$

( $\delta$ -Reduktion, li)  $\longrightarrow 42$

Anm.: Ratoren in rot, Randen in gold; li für linkest-innerst.

# Beispiel'

... $\delta$ -Reduktionsfolge: Applikative Ordnung

$$\begin{aligned} & (\lambda x. \lambda y. x * y) ((\lambda x. \lambda y. x + y) 9 5) 3 \\ & \quad (\beta\text{-Reduktion, li}) \longrightarrow (\lambda x. \lambda y. x * y) ((\lambda y. 9 + y) 5) 3 \\ & \quad (\beta\text{-Reduktion, li}) \longrightarrow (\lambda x. \lambda y. x * y) (9 + 5) 3 \\ & \quad (\delta\text{-Reduktion, li}) \longrightarrow (\lambda x. \lambda y. x * y) 14 3 \\ & \quad (\beta\text{-Reduktion, li}) \longrightarrow (\lambda y. 14 * y) 3 \\ & \quad (\beta\text{-Reduktion, li}) \longrightarrow 14 * 3 \\ & \quad (\delta\text{-Reduktion, li}) \longrightarrow 42 \end{aligned}$$

Anm.: Ratoren in rot, Randen in gold; li für linkest-innerst.

# Übungsaufgabe 12.4.1

## $\delta$ -Reduktionsfolgen

Gegeben ist der  $\lambda$ -Ausdruck  $e$ :

$$(\lambda x. \lambda. x * y) ((\lambda x. \lambda y. x + y) 9 5) 3$$

Ergänze die **applikative  $\delta$ -Reduktionsfolge** für diesen Ausdruck aus dem vorhergehenden Beispiel um

1. die **normale  $\delta$ -Reduktionsfolge**.
2. zwei **weitere** zur **Normalform** führende  **$\delta$ -Reduktionsfolgen** verschieden von der applikativen und normalen Folge.

Gibt es darüberhinaus weitere verschiedene zur Normalform führende Reduktionsfolgen für  $e$ ?



# Typisierte $\lambda$ -Kalküle

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

$3 : \text{Int}$

$(*) : \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$

$(\lambda x. 2 * x) : \text{Int} \rightarrow \text{Int}$

$(\lambda x. 2 * x) 3 : \text{Int}$

Dabei treten 2 Schwierigkeiten auf:

1. Ausdrücke mit Selbstanwendung (wie z.B. der Y-Kombinator) haben
  - keinen endlichen Typ, ihr Typ ist nicht durch einen gewöhnlichen endlichen Typausdruck beschreibbar.
2. Kombinatoren wie der Y-Kombinator können
  - nicht zur Modellierung von Rekursion verwendet werden.

# Überwindung

...der **Typisierungsschwierigkeit**:

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

...der **Rekursionschwierigkeit**:

- **Pragmatisch**: Explizite Hinzunahme der **Reduktionsregel**  
 $Y e \longrightarrow e (Y e)$   
zum Kalkül.

# Rechtfertigung, formale Fundierung

...des Umgangs mit **angereicherten angewandten  $\lambda$ -Kalkülen** u. des **pragmatischen** Hinzunehmens einer **Reduktionsregel** für Rekursion:

Resultate aus der **theoretischen Informatik**, insbesondere:

- ▶ Alonzo Church. **The Calculi of Lambda-Conversion**. Annals of Mathematical Studies, Vol. 6, Princeton University Press, 1941.

...u.a. zur Modellierung **ganzer Zahlen**, **Wahrheitswerten**, etc. durch Ausdrücke des **reinen  $\lambda$ -Kalküls**.

**Anmerkung:** Aus praktischer Sicht ist

- der Übergang zu **angewandten  $\lambda$ -Kalkülen** sinnvoll und nützlich (für theoretische Untersuchungen zur Berechenbarkeit (**Berechenbarkeitstheorie**) ist er kaum relevant).
- die Hinzunahme einer **Rekursionsregel** aus **Effizienzgründen** ebenfalls **zweckmäßig**.

# Kapitel 12.5

## Zusammenfassung

Vortrag V

Teil V

Kap. 12

12.1

12.2

12.3

12.4

**12.5**

12.6

Kap. 13

Kap. 14

Umgekehrte  
Klassen-  
zim-  
mer IV

Hinweis

Aufgabe

# Zusammenfassung aus Haskell-Perspektive

- Haskell beruht auf angewandten typisierten  $\lambda$ -Kalkülen.
- Übersetzer, Interpretierer prüfen, ob die Typisierung von Haskell-Programmen wohlgetypt, konsistent ist.
- Programmierer können Typdeklarationen angeben (aus-sagekräftigere Fehlermeldungen, Sicherheit), müssen aber nicht (bequem, doch u.U. mit unerwarteten Folgen, etwa bei “zufällig” korrekter, aber “ungemeinter” Typisierung: “gemeinte” Typisierung wäre bei Angabe bei der Typprüfung als inkonsistent aufgefallen).
- Typinformation (gleich ob angegeben oder nicht) wird vom Übersetzer, Interpretierer inferiert, berechnet.
- Rekursion kann unmittelbar ausgedrückt werden (Y-Kombinator nicht erforderlich).





# Schlussaneddote

...zur Entwicklung der  $\lambda$ -Notation anhand der mit einer anonymen  $\lambda$ -Abstraktion definierten Fakultätsfunktion:

```
fac :: Integer -> Integer
fac = \n -> (if n == 0 then 1 else (n * fac (n - 1)))
```

In Haskell abweichend vom  $\lambda$ -Kalkül also Verwendung von  
– “\” und “->” anstelle von “ $\lambda$ ” und “.”

...der Weg dorthin war kurvenreich:

-   $(\overbrace{n. n + 1})$  (Churchs handschriftliche Schreibweise)
-   $(\wedge n. n + 1)$  (Churchs notat. Zugeständnis an Schriftsetzer)
-   $(\lambda n. n + 1)$  (Pragmatische Umsetzung durch Schriftsetzer)
-   $(\backslash n \rightarrow n+1)$  (Approximative ASCII-Umsetzung in Haskell)

(siehe: Peter Pepper. Funktionale Programmierung in Opal, ML, Haskell und Gofer. Springer-V., 2. Auflage, 2003, S. 22.)

# Kapitel 12.6

## Leseempfehlungen

Vortrag V

Teil V

Kap. 12

12.1

12.2

12.3

12.4

12.5

**12.6**

Kap. 13






Kap. 14

Umgekehrte  
Klassen-  
zim-  
mer IV

Hinweis




Aufgabe

# Basiseleseempfehlungen für Kapitel 12 (1)




-  Henrik P. Barendregt, Wil Dekkers, Richard Statman. *Lambda Calculus with Types*. Cambridge University Press, 2012.
-  Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Kapitel 19, Berechenbarkeit und Lambda-Kalkül)
-  Antonie J.T. Davie. *An Introduction to Functional Programming Systems using Haskell*. Cambridge University Press, 1992. (Kapitel 5, Lambda Calculus)
-  Martin Erwig. *Grundlagen funktionaler Programmierung*. Oldenbourg Verlag, 1999. (Kapitel 4, Der Lambda-Kalkül)
-  Anthony J. Field, Peter G. Robinson. *Functional Programming*. Addison-Wesley, 1988. (Kapitel 6, Mathematical foundations: the lambda calculus)





## Basiseleseempfehlungen für Kapitel 12 (2)

-  Chris Hankin. *An Introduction to Lambda Calculi for Computer Scientists*. King's College London Publications, 2004. (Kapitel 1, Introduction; Kapitel 2, Notation and the Basic Theory; Kapitel 3, Reduction; Kapitel 10, Further Reading)
-  Greg Michaelson. *An Introduction to Functional Programming through Lambda Calculus*. Dover Publications, 2. Auflage, 2011. (Kapitel 2, Lambda calculus; Kapitel 4.1, Repetition, iteration and recursion; Kapitel 4.3, Passing a function to itself; Kapitel 4.6, Recursion notation; Kapitel 8, Evaluation)
-  Ingo Wegener. *Grenzen der Berechenbarkeit*. In Informatik-Handbuch, Peter Rechenberg, Gustav Pomberger (Hrsg.), Carl Hanser Verlag, 4. Auflage, 111-118, 2006. (Kapitel 4.1, Rechnermodelle und die Churchsche These)







# Weiterführ. Leseempfehlungen für Kap. 12 (1)

-  Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, Philip Wadler. *The Call-by-Need Lambda Calculus*. In Conference Record of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95), 233-246, 1995.
-  Hendrik Pieter Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Revised Edn., North-Holland, 1984. (Kapitel 1, Introduction; Kapitel 2, Conversion; Kapitel 3, Reduction; Kapitel 6, Classical Lambda Calculus; Kapitel 11, Fundamental Theorems)
-  Gordon Plotkin. *Call-by-name, Call-by-value, and the  $\lambda$ -Calculus*. Theoretical Computer Science 1:125-159, 1975.






## Weiterführ. Leseempfehlungen für Kap. 12 (2)

-  Achim Jung. *Berechnungsmodelle*. In Informatik-Handbuch, Peter Rechenberg, Gustav Pomberger (Hrsg.), Carl Hanser Verlag, 4. Auflage, 73-88, 2006. (Kapitel 2.1, Speicherorientierte Modelle: Turing-Maschinen, Registermaschinen; Kapitel 2.2, Funktionale Modelle: Algebraische Kombinationen, Primitive Rekursion,  $\mu$ -Rekursion,  $\lambda$ -Kalkül)
-  Allen B. Tucker (Editor-in-Chief). *Computer Science Handbook*. Chapman & Hall/CRC, 2004. (Kapitel 92.3, The Lambda Calculus: Foundation of All Functional Languages)



## Leseempfh. f. Kap. 12: Die frühen Arbeiten

-  Wilhelm Ackermann. *Zum Hilbertschen Aufbau der reellen Zahlen*. *Mathematische Annalen* 99:118-133, 1928.
-  Alonzo Church. *The Calculi of Lambda-Conversion*. *Annals of Mathematical Studies*, Vol. 6, Princeton University Press, 1941.
-  Stephen C. Kleene. *General Recursive Functions of Natural Numbers*. *Mathematische Annalen* 112:727-742, 1936.
-  Stephen C. Kleene.  *$\lambda$ -Definability and Recursiveness*. *Duke Mathematical Journal* 2:340-352, 1936.
-  Rózsa Péter. *Über den Zusammenhang der verschiedenen Begriffe der rekursiven Funktionen*. *Mathematische Annalen* 110:612-632, 1934.
-  Rózsa Péter. *Konstruktion nichtrekursiver Funktionen*. *Mathematische Annalen* 111:42-60, 1935.

# Leseempfh. f. Kap. 12: Über die frühen Arbeiten

-  Stephen C. Kleene. *Origins of Recursive Function Theory*. Annals of the History of Computing 3:52-67, 1981.
-  Robin Gandy. *The Confluence of Ideas in 1936*. In Rolf Herken (Hrsg.), *The Universal Turing Machine: A Half-Century Survey*. Springer-V., 2. Auflage, 51-102, 1995.
-  William Newman. *Alan Turing Remembered – A Unique Firsthand Account of Formative Experiences with Alan Turing*. Communications of the ACM 55(12):39-41, 2012.
-  Robert M. French. *Moving Beyond the Turing Test*. Communications of the ACM 55(12):74-77, 2012.
-  Omer Reingold. *Through the Lens of a Passionate Theoretician*. Communications of the ACM 63(3):25-27, 2020.

# Leseempf. f. Kap. 12: Über die frühen Arbeiten

-  Uwe Schöning, Wolfgang Thomas. *Turings Arbeiten über Berechenbarkeit – eine Einführung und Lesehilfe*. Informatik Spektrum 35(4):253-260, 2012.
-  Boris A. Trakhtenbrot *Comparing the Church and Turing Approaches: Two Prophetic Messages*. In Rolf Herken (Hrsg.), *The Universal Turing Machine: A Half-Century Survey*. Springer-V., 2. Auflage, 557-582, 1995.

Vortrag V

Teil V

Kap. 12

12.1

12.2

12.3

12.4

12.5

12.6

Kap. 13

Kap. 14

Umgekehrt




Klassen-  
zimmer IV

Hinweis

Aufgabe

# Leseempfh. f. Kap. 12: Über das Wesen

...den Einfluss und Geist von Berechnung.

-  Ian Horswill. *What is Computation?* Crossroads, the ACM Magazine for Students 18(3):8-14, 2012.
-  Avi Wigderson. *Mathematics and Computation: A Theory Revolutionizing Technology and Science*. Princeton University Press, 2019.
-  David Harel. *Algorithmics: The Spirit of Computing*. Addison-Wesley, 2. Auflage, 1992.

# Kapitel 13

## Auswertungsordnungen

Vortrag V

Teil V

Kap. 12

**Kap. 13**

13.1

13.2

13.3

13.4

13.5

13.6

13.7

13.9

Kap. 14

Umgekehrt

Klassen-  
zim-  
mer IV

Hinweis

Aufgabe



# Kapitel 13.1

## Überblick, Orientierung

Vortrag V

Teil V

Kap. 12

Kap. 13

**13.1**

13.2

13.3

13.4

13.5

13.6

13.7

13.9

Kap. 14

Umgekehrt

Klassen-  
zim-  
mer IV

Hinweis

Aufgabe

# Auswertungsordnungen

...legen fest:

```
2^3+fac(fib(squ(2+2)))+3*5+fib(fac(7*(5+3)))+fib((5+7)*2)
->> ... ->> 'große Zahl'
```

1. **Wo** wird in einem komplexen Ausdruck 'gerechnet'?

Mögliche Antworten:

- Links
- Rechts
- Halblinks
- Mittig
- ...

2. **Wann** werden Funktionstermargumente ausgewertet?

Mögliche Antworten:

- Vor
- Nach
- Mal so, mal so

der Expansion des Funktionsterms.

# Die Antwort

...auf:

– Frage 1 ist pragmatisch:

– So weit links wie möglich!

$2^3 + \text{fac}(\text{fib}(\text{squ}(2+2))) + 3 * 5 + \text{fib}(\text{fac}(7 * (5+3))) + \dots$   
->>  $8 + \text{fac}(\text{fib}(\text{squ}(2+2))) + 3 * 5 + \text{fib}(\text{fac}(7 * (5+3))) + \dots$

– Frage 2 ist spannender und geteilt:

– Applikativ: Argumentauswertung vor Expansion

– Normal: Argumentauswertung nach Expansion

Applikativ:

$\text{squ}(2+2) \xrightarrow{\text{Simpl.}} \text{squ } 4 \xrightarrow{\text{Exp.}} 4 * 4 \xrightarrow{\text{Simpl.}} 16$

Normal:

$\text{squ}(2+2) \xrightarrow{\text{Exp.}} (2+2) * (2+2) \xrightarrow{\text{Simpl.}} 4 * (2+2) \xrightarrow{\text{Simpl.}} 4 * 4 \xrightarrow{\text{Simpl.}} 16$

# Somit: Wir unterscheiden

...zwei Auswertungsordnungen als Antwort auf Frage 2:

## 1. Applikativ

- Kennzeichen: Arg.Auswertung vor Expansion, d.h. sofortige, frühe Arg.Auswertung in Funktionstermen.

## 2. Normal

- Kennzeichen: Arg.Auswertung nach Expansion, d.h. aufgeschobene, späte Arg.Auswertung in Funktionstermen.

mit drei Operationalisierungen als Antwort auf Frage 1:

1. Linksapplikativ: Frühe, fleißige Auswertung (engl. eager evaluation).
2. Linksnormal (ohne implementierungspraktische Bedeutung).
3. Linksnormal mit Ausdrucksteilung (engl. expression sharing) als Implementierungskniff: Späte, faule Auswertung (engl. lazy evaluation).

# In einem Satz

...Auswertungsordnungen geben eine konkrete Antwort auf Frage 1 und Frage 2 und organisieren damit das Zusammenspiel des

- Expandierens (E) (von Funktionstermen)
- Simplifizierens (S) (von Ausdrücken verschieden von Funktionstermen)

von Ausdrücken mit dem Ziel, sie so weit zu vereinfachen wie irgend möglich, also ihren Wert zu berechnen:

```
2^3+fac(fib(squ(2+2)))+3*5+fib(fac(7*(5+3)))+fib((5+7)*2)
->> ... ->> 'große Zahl'
```

# Kapitel 13.2

## Applikative, normale Funktionstermauswertung

Vortrag V

Teil V

Kap. 12

Kap. 13

13.1

**13.2**

13.2.1

13.2.2

13.2.3

13.3

13.4

13.5

13.6

13.7

13.9

Kap. 14

Umgekehrt

Klassen-  
zimmer  
IV

Hinweis

Aufgabe

# Kapitel 13.2.1

## Applikative Funktionstermauswertung

Vortrag V

Teil V

Kap. 12

Kap. 13

13.1

13.2

**13.2.1**

13.2.2

13.2.3

13.3

13.4

13.5

13.6

13.7

13.9

Kap. 14

Umgekehrt

Klassen-  
zimmer  
IV

Hinweis

Aufgabe

# Applikative Funktionstermauswertung

...heißt **Argumentauswertung vor Expansion**:

Ein Funktionsterm ( $f \text{ ausd}_1 \dots \text{ ausd}_n$ ) wird ausgewertet, indem:

1. die Argumentausdrücke  $\text{ausd}_1, \dots, \text{ausd}_n$  werden vollständig zu ihren Werten  $w_1, \dots, w_n$  ausgewertet.
2.  $w_1, \dots, w_n$  werden im Rumpf von  $f$  für die Parameter von  $f$  eingesetzt.
3. der entstandene expandierte Ausdruck wird ausgewertet.

Das Vorgehen bei **applikativer Argumentauswertung** motiviert **zusätzliche Sprechweisen** (s. auch Kap. 13.5):

- Wertparameter-, innerste, strikte Auswertung (engl. *call-by-value, innermost, strict evaluation*).



# Kapitel 13.2.2

## Normale Funktionstermauswertung

Vortrag V

Teil V

Kap. 12

Kap. 13

13.1

13.2

13.2.1

**13.2.2**

13.2.3

13.3

13.4

13.5

13.6

13.7

13.9

Kap. 14

Umgekehrt

Klassen-  
zimmer  
IV

Hinweis

Aufgabe

# Normale Funktionstermauswertung

...heißt **Argumentauswertung nach Expansion**:

Ein Funktionsterm  $(f \text{ ausd}_1 \dots \text{ausd}_n)$  wird ausgewertet, indem:

1. die Argumentausdrücke  $\text{ausd}_1, \dots, \text{ausd}_n$  werden unausgewertet im Rumpf von  $f$  für die Parameter von  $f$  eingesetzt.
2. der entstandene expandierte Ausdruck wird ausgewertet.

Das Vorgehen **normaler Argumentauswertung** motiviert gleichfalls **zusätzlich Sprechweisen** (s. auch Kap. 13.5):

- Namensparameter-, äußerste Auswertung (engl. **call-by-name, outermost evaluation**).

# Kapitel 13.2.3

## Beispiele

Vortrag V

Teil V

Kap. 12

Kap. 13

13.1

13.2

13.2.1

13.2.2

**13.2.3**

13.3

13.4

13.5

13.6

13.7

13.9

Kap. 14

Umgekehrte  
Klassen-  
zimmer  
IV

Hinweis

Aufgabe

# Die Funktion binom3

```
binom3 x y :: Int -> Int -> Int
```

```
binom3 x y = (x + y) * (x - y)
```

```
binom3 16 ((5-3)*7) ->> ... ->> 60
```

## Applikative Funktionstermauswertung:

```
binom3 16 ((5-3)*7) (erst Arg. vereinfachen...)
```

```
(S) ->> binom3 16 (2*7)
```

```
(S) ->> binom3 16 14 (...dann expandieren!)
```

```
(E) ->> (16 + 14) * (16 - 14)
```

```
(S) ->> ...
```

```
(S) ->> 60
```

## Normale Funktionstermauswertung:

```
binom3 16 ((5-3)*7) (sofort expandieren!)
```

```
(E) ->> (16 + ((5-3)*7)) * (16 - ((5-3)*7))
```

```
(S) ->> ...
```

```
(S) ->> 60
```

# Die Fakultätsfunktion fac

```
fac :: Integer -> Integer
fac n = if n == 0 then 1 else (n * fac (n - 1))
fac 2 ->> ... ->> 2
```

fac 2

- (E) ->> if 2 == 0 then 1 else (2 \* fac (2 - 1))
- (S) ->> if False then 1 else (2 \* fac (2 - 1))
- (S) ->> 2 \* (fac (2 - 1))
- (?) ...**expandierend** oder **simplifizierend** fortfahren?

Mit dem Funktionsterm (fac (2 - 1)) kann jetzt:

- **applikativ** (**simplifizierend** mit Rechnen auf Arg.-Position)
- **normal** (mit **Expandieren** des Aufrufs)

auswertend fortfahren werden.

...beide Möglichkeiten führen wir im Detail aus.

# Applikative, normale Auswertungsweise

**Applikativ:**  $2 * \text{fac } (2-1)$  (Arg.vereinf. zuerst)  
(S)  $\rightarrow 2 * \text{fac } 1$   
(E)  $\rightarrow 2 * (\text{if } 1 == 0 \text{ then } 1 \text{ else } (1 * \text{fac } (1-1)))$   
(S)  $\rightarrow 2 * (\text{if } \text{False} \text{ then } 1 \text{ else } (1 * \text{fac } (1-1)))$   
(S)  $\rightarrow 2 * (1 * \text{fac } (1-1))$   
(S)  $\rightarrow \dots$ in diesem Stil fortfahren.

**Normal:**  $2 * \text{fac } (2-1)$  (Expansion zuerst)  
(E)  $\rightarrow 2 * (\text{if } (2-1) == 0 \text{ then } 1$   
 $\quad \quad \quad \text{else } ((2-1) * \text{fac } ((2-1)-1)))$   
(S)  $\rightarrow 2 * (\text{if } 1 == 0 \text{ then } 1$   
 $\quad \quad \quad \text{else } ((2-1) * \text{fac } ((2-1)-1)))$   
(S)  $\rightarrow 2 * (\text{if } \text{False} \text{ then } 1$   
 $\quad \quad \quad \text{else } ((2-1) * \text{fac } ((2-1)-1)))$   
(S)  $\rightarrow 2 * ((2-1) * \text{fac } ((2-1)-1))$   
(S)  $\rightarrow 2 * (1 * \text{fac } ((2-1)-1))$   
(E)  $\rightarrow \dots$ in diesem Stil fortfahren.

# Die vollständige applikative Auswertung (1)

fac n = if n == 0 then 1 else (n \* fac (n - 1))  
...für den Aufruf fac (1+1) (statt fac 2):

```
      fac (1+1)           (Argument wird sofort ausgewertet)
(S) ->> fac 2           (Expansion nach max. Argumentvereinf.)
(E) ->> if 2 == 0 then 1 else (2 * fac (2-1))
(S) ->> if False then 1 else (2 * fac (2-1))
(S) ->> 2 * fac (2-1)   (Arg. wird sofort ausgewertet)
(S) ->> 2 * fac 1       (Exp. nach max. Argumentvereinf.)
(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)) (Arg. wird sofort ausgewertet)
(S) ->> 2 * (1 * fac 0)   (Exp. nach max. Arg.vereinf.)
(E) ->> 2 * (1 * (if 0 == 0 then 1
                  else (0 * fac (0-1))))
```

# Die vollständige applikative Auswertung (2)

```
(S) ->> 2 * (1 * (if False then 1  
                  else (0 * fac (0-1))))
```

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

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

```
(S) ->> 2
```

↪ Applikative, fleißige, frühe Argumentauswertung  
(engl. applicative, eager order evaluation)



# Die vollständige normale Auswertung (1)

fac n = if n == 0 then 1 else (n \* fac (n - 1))  
...für den Aufruf fac (1+1) (statt fac 2):

fac (1+1) (Sofortige Exp., keine vorh. Arg.vereinf.)

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

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

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

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

(S) ->> (2 \* fac ((1+1)-1)) (Sofortige Exp.)

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

(3S) ->> 2 \* (if False then 1  
              else (((1+1)-1) \* fac (((1+1)-1)-1)))

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

# Die vollständige normale Auswertung (2)

(S)  $\rightarrow$  2 \* ((2-1) \* fac (((1+1)-1)-1))

(S)  $\rightarrow$  2 \* (1 \* fac (((1+1)-1)-1)) - Sofortige Exp.

(E)  $\rightarrow$  2 \* (1 \* (if (((1+1)-1)-1) == 0 then 1  
else (((1+1)-1)-1) \* fac (((1+1)-1)-1)))

(4S)  $\rightarrow$  2 \* (1 \* (if True then 1  
else (((1+1)-1)-1) \* fac (((1+1)-1)-1)))

(S)  $\rightarrow$  2 \* (1 \* 1)

(S)  $\rightarrow$  2 \* 1

(S)  $\rightarrow$  2

$\rightsquigarrow$  Normale Argumentauswertung  
(engl. normal order evaluation)

# Kapitel 13.3

## Linksapplikative, linksnormale Auswertung

# Linksapplikative, linksnormale Auswertung

...wichtige praktische **Operationalisierungen**

- applikativer
- normaler

Auswertung, die neben der Art der

- ▶ Argumentauswertung von Funktionstermen (**vor/nach Expansion**)

auch festlegen

- ▶ an welcher Stelle in einem Ausdruck gerechnet wird (**linkstmöglich**)

und damit eine **eindeutige Auswertungsreihenfolge** für **komplexe Ausdrücke** wie:

$2^3 + \text{fac}(\text{fib}(\text{squ}(2+2))) + 3 * 5 + \text{fib}(\text{fac}(7 * (5+3))) + \text{fib}((5+7) * 2)$

festlegen.

# Im Detail: Applikativ, normal auszuwerten

...sind Antwort auf die **erste operationell wichtige Frage** für die Auswertung komplexer Ausdrücke:

## 1. **Wie** ist mit (Funktions-) Argumenten umzugehen?

↪ Ausgewertet oder unausgewertet übergeben?

- **Applikativ (innerst)**: Ausgewertet übergeben.
- **Normal (äußerst)**: Unausgewertet übergeben.

**Linksapplikativ, linksnormal** auszuwerten sind Antwort auf die **zweite operationell wichtige Frage**:

## 2. **Wo** ist im Ausdruck auszuwerten?

↪ Links, rechts, halblinks, in der Mitte?

- **Linksapplikativ (linksinnerst)**: Linkstinnerste Stelle.
- **Linksnormal (linksäußerst)**: Linkstäußerste Stelle.

$2^3 + \text{fac}(\text{fib}(\text{squ}(2+2))) + 3*5 + \text{fib}(\text{fac}(7*(5+3))) + \text{fib}((5+7)*2)$

# Beispiel: 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 +...
->> ...
```

# Beispiel: 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 +...
->> ...
```

# Beachte

...dass bei der Expansion von `fib` bei **linksnormaler** Auswertung statt der **musterbasierten** Definition von `fib`:

```
fib :: Int -> Int      -- Musterbasierte Definition
fib 0 = 0
fib 1 = 1
fib n = fib (n-2) + fib (n-1)
```

für dieses Beispiel notationell einfacher die Version mit **geschichtetem Fallunterscheidungsausdruck** verwendet worden ist:

```
fib :: Int -> Int  -- Geschachtelte Falluntersch.
fib n = if n==0 then 0
        else if n==1 then 1
            else fib (n-2) + fib (n-1)
```



# Naheliegende Fragen

Welche Auswirkungen hat die Wahl von (links-) applikativer oder (links-) normaler Auswertungsordnung?

...auf:

1. Terminierungsverh., -häufigk.? (Th. 13.3.2, Th. 13.3.3)
2. Terminierungsgeschwindigkeit? (Th. 13.3.3, Kap. 13.5)
3. berechneten Ausdruckswert im Term.fall? (Th. 13.3.1)

# Die in Kapitel 12 bereits gegebenen Antworten

Die Church/Rosser-Theoreme 12.3.4.1, 12.3.4.2 garantieren:

## Theorem 13.3.1 (Wertdeterminiertheit)

Jede terminierende Auswertungsfolge für einen Ausdruck endet mit demselben Ergebnis.

**Informell:** Terminierende Auswertungsfolgen widersprechen sich nicht.

## Theorem 13.3.2 (Terminierungshäufigkeit)

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

**Informell:** (Links-) normale Auswertung terminiert am häufigsten.

# Die in Kap. 12 bereits gegeb. Antworten (fgs.)

## Theorem 13.3.3 (Abweichendes Terminierungsverh.)

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

**Summa summarum:** (Links-) applikative und (links-) normale Auswertungsordnung können sich für einen Ausdruck unterscheiden in:

- ▶ Terminierungsverhalten (d.h. Terminierungshäufigkeit)
- ▶ Terminierungsgeschwindigkeit (d.h. Performanz)

nicht aber im:

- ▶ Ergebnis (wenn beide terminieren)

# Beispiele für den Beleg unterschiedlicher

...Terminierungsgeschwindigkeit, -häufigkeit:

- Die Quadratfunktion `squ` auf ganzen Zahlen:

```
squ :: Integer -> Integer
```

```
squ n = n * n
```

- Die nichtterminierende Inkrementfunktion `infinite`:

```
infinite :: Integer
```

```
infinite = 1 + infinite
```

- Die Projektionsfunktion `fst` für Paare:

```
fst :: (a,b) -> a
```

```
fst (x,_) = x
```

- Drei Ausdrücke `a1`, `a2`, `a3`:

```
a1 = (17+4) + squ (squ (squ (1+1))) + (2*11)
```

```
a2 = fst (2*21,squ (squ (squ (1+1))))
```

```
a3 = fst (2*21,infinite)
```

# Linksapplikative, linkestinnerste (LI) Auswert.

...von  $a_1 = (17+4) + \text{squ}(\text{squ}(\text{squ}(1+1))) + (2*11)$ :

$$((17+4) + \text{squ}(\text{squ}(\text{squ}(1+1)))) + (2*11)$$

$$\text{(LI-S)} \rightarrow (21 + \text{squ}(\text{squ}(\text{squ}(1+1)))) + (2*11)$$

$$\text{(LI-S)} \rightarrow (21 + \text{squ}(\text{squ}(\text{squ } 2))) + (2*11)$$

$$\text{(LI-E)} \rightarrow (21 + \text{squ}(\text{squ}(2*2))) + (2*11)$$

$$\text{(LI-S)} \rightarrow (21 + \text{squ}(\text{squ } 4)) + (2*11)$$

$$\text{(LI-E)} \rightarrow (21 + \text{squ}(4*4)) + (2*11)$$

$$\text{(LI-S)} \rightarrow (21 + \text{squ } 16) + (2*11)$$

$$\text{(LI-E)} \rightarrow (21 + 16*16) + (2*11)$$

$$\text{(LI-S)} \rightarrow (21 + 256) + (2*11)$$

$$\text{(LI-S)} \rightarrow 277 + (2*11)$$

$$\text{(LI-S)} \rightarrow 277 + 22$$

$$\text{(LI-S)} \rightarrow 299$$

Insgesamt:  $1 + 7 + 3 = 11$  Schritte

...davon 7 Schritte für  $\text{squ}(\text{squ}(\text{squ}(1+1)))$

# Linksnormale, linkestäußerste (LÄ) Auswert.

...von  $a_1 = (17+4) + \text{squ}(\text{squ}(\text{squ}(1+1))) + (2*11)$ :

$((17+4) + \text{squ}(\text{squ}(\text{squ}(1+1)))) + (2*11)$

(LÄ-S)  $\rightarrow$   $(21 + \text{squ}(\text{squ}(\text{squ}(1+1)))) + (2*11)$

(LÄ-E)  $\rightarrow$   $(21 + \text{squ}(\text{squ}(1+1)) * \text{squ}(\text{squ}(1+1))) + (2*11)$

(LÄ-E)  $\rightarrow$   $(21 + ((\text{squ}(1+1)) * (\text{squ}(1+1))) * \text{squ}(\text{squ}(1+1))) + (2*11)$

(LÄ-E)  $\rightarrow$   $(21 + ((1+1) * (1+1) * \text{sq}(1+1)) * \text{squ}(\text{squ}(1+1))) + (2*11)$

(LÄ-S)  $\rightarrow$   $(21 + (2*(1+1)*\text{squ}(1+1)) * \text{squ}(\text{squ}(1+1))) + (2*11)$

(LÄ-S)  $\rightarrow$   $(21 + (2*2*\text{squ}(1+1)) * \text{squ}(\text{squ}(1+1))) + (2*11)$

(LÄ-S)  $\rightarrow$   $(21 + (4 * \text{squ}(1+1)) * \text{squ}(\text{squ}(1+1))) + (2*11)$

(LÄ-E)  $\rightarrow$   $(21 + (4*((1+1)*(1+1))) * \text{squ}(\text{squ}(1+1))) + (2*11)$

(LÄ-S)  $\rightarrow$   $(21 + (4*(2*(1+1))) * \text{squ}(\text{squ}(1+1))) + (2*11)$

(LÄ-S)  $\rightarrow$   $(21 + (4*(2*2)) * \text{squ}(\text{squ}(1+1))) + (2*11)$

(LÄ-S)  $\rightarrow$   $(21 + (4*4) * \text{squ}(\text{squ}(1+1))) + (2*11)$

(LÄ-S)  $\rightarrow$   $(21 + 16 * \text{squ}(\text{squ}(1+1))) + (2*11)$

$\rightarrow$  ...

# Linksnormale, linkestäußerste (LÄ) Ausw. (fgs.)

->> ...

$$(L\ddot{A}-S) \rightarrow (21 + (16 * 16)) + (2*11)$$

$$(L\ddot{A}-S) \rightarrow (21 + 256) + (2*11)$$

$$(L\ddot{A}-S) \rightarrow 277 + (2*11)$$

$$(L\ddot{A}-S) \rightarrow 277 + 22$$

$$(L\ddot{A}-S) \rightarrow 299$$

Insgesamt:  $1 + ((1+10)+(1+10)+1) + 3 = 27$  Schritte

...davon 23 Schritte für `squ (squ (squ (1+1)))`

# Linksapplikativ effizienter als linksnormal?

Nicht immer! Betrachte den zweiten Ausdruck:

```
a2 = first (2*21, squ (squ (squ (1+1))))
```

## ► Linksapplikative Auswertung:

```
                first (2*21, squ (squ (squ (1+1))))  
(LI-S) ->> first (42, squ (squ (squ (1+1))))  
          ->> ...  
(LI-S) ->> first (42, 256)  
(LI-E) ->> 42
```

Insgesamt:  $1+7+1=9$  Schritte (davon 7 für den Wert des unbenötigten zweiten Arguments!)

## ► Linksnormale Auswertung:

```
                first (2*21, squ (squ (squ (1+1))))  
(LÄ-E) ->> 2*21  
(LÄ-S) ->> 42
```

Insgesamt: 2 Schritte (das unbenötigte zweite Argument wird überhaupt nicht ausgewertet!)



# Linksapplikativ, linksnormal 'termin.-gleich'?

Nicht immer! Betrachte den dritten Ausdruck:

```
a3 = first (2*21,infinite)
```

## ► Linksapplikative Auswertung:

```
first (2*21,infinite)
```

```
(LI-S) ->> first (42,infinite)
```

```
(LI-E) ->> first (42,1+infinite)
```

```
(LI-E) ->> first (42,1+(1+infinite))
```

```
(LI-E) ->> ...
```

```
(LI-E) ->> first (42,1+(1+(1+(...+(1+infinite)...)))
```

```
(LI-E) ->> ...
```

Insgesamt: Nichtterminierung, kein Resultat: **undefiniert!**

## ► Linksnormale Auswertung:

```
first (2*21,infinite)
```

```
(LÄ-E) ->> 2*21
```

```
(LÄ-S) ->> 42
```

Insgesamt: Terminierung, Res. 42 nach 2 Schritten: **def.!**

# Unterschiedliche Terminierungshäufigkeit

..von linksapplikativer und linksnormaler Auswertung.

Das Beispiel des zweiten Ausdrucks

```
a2 = first (2*21,squ (squ (squ (1+1))))
```

zeigt:

- Linksapplikative und linksnormale Auswertung können sich im Terminierungsverhalten unterscheiden:
  - Applikativ: **Nichttermination**, kein Resultat: **undefiniert**.
  - Normal: **Termination**, sehr wohl ein Resultat: **definiert**.

**Wichtig:** Die umgekehrte Situation ist nicht möglich (siehe Theorem 13.3.1)!

# Unterschiedliche Terminierungsgeschwindigkeit

...von linksapplikativer und linksnormaler Auswertung.

Das Beispiel des ersten Ausdrucks

$$a1 = (17+4) + \text{squ} (\text{squ} (\text{squ} (1+1)))) + (2*11)$$

zeigt:

- ▶ **Ergebnisgleichheit:** Terminieren linksapplikative und linksnormale Auswertungsordnung für einen Ausdruck beide, so terminieren sie mit demselben Resultat (s. Theorem 13.3.1).
- ▶ **Schritzzahlungleichheit:** Linksapplikative und linksnormale Auswertung können bis zur Terminierung (mit gleichem Endresultat) unterschiedlich viele Expansions- und Simplifikationsschritte benötigen.

# Unordenbare Performanz

..von **linksapplikativer** und **linksnormaler** Auswertung.

Die Beispiele aller drei **Ausdrücke** zusammen zeigen:

$$a1 = (17+4) + \text{squ} (\text{squ} (\text{squ} (1+1)))) + (2*11)$$

$$a2 = \text{fst} (2*21, \text{squ} (\text{squ} (\text{squ} (1+1))))$$

$$a3 = \text{fst} (2*21, \text{infinite})$$

dass weder **linksapplikative** noch **linksnormale** Auswertung der jeweils anderen Auswertungsordnung stets überlegen ist.

# Kapitel 13.4

## Späte, faule Auswertung: Eine Frage der Implementierung

# Linksnormale Auswertung

...hat ein **Effizienzproblem**:

- ▶ Funktionstermargumente werden (im schlimmsten Fall) so oft ausgewertet wie sie im Rumpf vorkommen.

Zum Vergleich **linksapplikative Auswertung**:

- ▶ Funktionstermargumente werden genau einmal ausgewertet, unabhängig davon, wie oft sie im Rumpf vorkommen.

**Ziel:**

- ▶ Linksnormale Auswertung so zu implementieren, dass Mehrfachauswertungen von Ausdrücken vermieden werden

und die **Effizienzlücke** im Vergleich zu linksapplikativer Auswertung geschlossen wird.

# Dafür gebraucht: Ein Implementierungskniff!

...damit Mehrfachauswertungen von Ausdrücken vermieden werden.

## Implementierungskniff:

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

## Damit:

- ▶ Ausdrucksauswertungen werden zu **Transformationen auf Graphen**.
- ▶ Wird ein Ausdruck ausgewertet, steht sein Wert an allen Verwendungsstellen zur Verfügung.

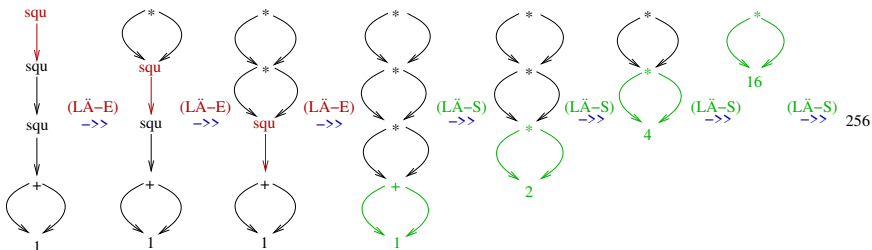
## Name der resultierenden Auswertungsordnung:

- ▶ **Späte, faule Auswertung** (engl. **lazy evaluation**)!

# Späte, faule Auswertung

...garantiert, dass Argumente **höchstens einmal** ausgewertet werden (möglicherweise also **gar nicht!**).

**Veranschaulichung:** Ausdrucksrepräsentation, Ausdrucksauswertung auf **Graphen**:



Insgesamt: **7 Schritte.**

(Statt **22 Schritte** bei (naiver) **linksnormaler** Auswertung.)



# Zusammengefasst

## Späte, faule Auswertung (engl. lazy evaluation)

- ▶ ist eine **effiziente** Implementierungsumsetzung **linksnormaler** Auswertung.
- ▶ erfordert implementierungstechnisch eine Darstellung von Ausdrücken in Form von Graphen und Graphtransformationen zu ihrer Auswertung.
- ▶ ist 'vergleichbar' performant wie **frühe, fleißige Auswertung** (engl. **eager evaluation**), wenn alle Argumente benötigt werden.

## Insgesamt: Späte, faule Auswertung

- ▶ erreicht annähernd den Vorteil **applikativer Auswertung** (**Effizienz!**), ohne dafür Abstriche am Vorteil **normaler Auswertung** (**Terminierungshäufigkeit!**) vornehmen zu müssen.
- ▶ vereint damit möglichst gut die Vorteile beider.

# Kapitel 13.5

## Zusätzliche Charakterisierungen der Auswertungsordnungen

Vortrag V

Teil V

Kap. 12

Kap. 13

13.1

13.2

13.3

13.4

**13.5**

13.4.1

13.5.2

13.5.3

13.5.4

13.5.5

13.6

13.7

13.9

Kap. 14

Umgekehrte  
Klassen-  
zim-  
mer IV

Hinweis

Aufgabe

# Zusätzliche Charakterisierungen

...der **Auswertungsordnungen** über **Analogien** bzw. **Betrachtungen** zu:

1. Parameterübergabemechanismen
2. Auswertungspositionen
3. Argumentauswertungshäufigkeiten
4. Definiertheitszusammenhang von Argument und Funktion

# Kapitel 13.4.1

## Über Parameterübergabemechanismen

# Parameterübergabemechanismen analogien

Applikative Auswertungsordnung entspricht

- ▶ Call-by-**value**

Normale Auswertungsordnung entspricht

- ▶ Call-by-**name**

Späte Auswertungsordnung entspricht

- ▶ Call-by-**need**

# Kapitel 13.5.2

## Über Auswertungspositionen

Vortrag V

Teil V

Kap. 12

Kap. 13

13.1

13.2

13.3

13.4

13.5

13.4.1

**13.5.2**

13.5.3

13.5.4

13.5.5

13.6

13.7

13.9

Kap. 14

Umgekehrte  
Klassen-  
zim-  
mer IV

Hinweis

Aufgabe

# Auswertungspositionen

## Applikative Auswertungsordnung

- ▶ **Innerste Auswertungsordnung:** Reduziere nur Redexe, die keine Redexe enthalten.
- ▶ **Linksapplikative, linkestinnerste Auswertungsordnung:** Reduziere stets den linkesten innersten Redex, der keine Redexe enthält.  
**Frühe, fleißige Auswertungsordn.** (engl. **eager evaluation**).

## Normale Auswertungsordnung

- ▶ **Äußerste Auswertungsordnung:** Reduziere nur Redexe, die nicht in anderen Redexen enthalten sind.
- ▶ **Linksnormale, linkestäußerste Auswertungsordnung:** Reduziere stets den linkesten äußersten Redex, der nicht in anderen Redexen enthalten ist.  
**Späte, faule Auswertungsordnung** (engl. **lazy evaluation**) bei effizienter Implementierung (s. **Kap. 13.4**).

# Kapitel 13.5.3

## Über Argumentauswertungshäufigkeit

Vortrag V

Teil V

Kap. 12

Kap. 13

13.1

13.2

13.3

13.4

13.5

13.4.1

13.5.2

**13.5.3**

13.5.4

13.5.5

13.6

13.7

13.9

Kap. 14

Umgekehr

Klassen-

zim-

mer IV

Hinweis

Aufgabe



# Argumentauswertungshäufigkeit

## 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.

...späte, faule Auswertung ist damit am effizientesten, benötigt i.a. aber mehr Speicher und hat Zusatzaufwand für die Verwaltung geteilter Datenstrukturen.

# Beispiel

Betrachte die **Funktion**:

```
f :: Int -> Int -> Int -> Int
f x y z = if x>42 then y+y else z^z
```

und den **Aufruf**:

```
f 45 (squ (5*(2+3))) (squ ((2{+3}*7))
```

# Applikative Auswertung von f

`f x y z = if x>42 then y+y else z^z`

Applikative Auswertung:

```
      f 45 (squ (5*(2+3))) (squ ((2+3)*7))
(2S) ->> f 45 (squ (5*5)) (squ (5*7))
(2S) ->> f 45 (squ 25) (squ 35)
(2E) ->> f 45 (25*25) (35*35)
(2S) ->> f 45 625 1225
(E) ->> if 45>42 then 625+625 else 1125^1125
(S) ->> if True then 625+625 else 1125^1125
(S) ->> 625+625
(S) ->> 1250
```

...die Argumente `(squ (5*(2+3)))` und `(squ ((2+3)*7))` werden beide **genau einmal** ausgewertet (ohne dass der Wert von `(squ ((2+3)*7))` benötigt wird).

# Normale Auswertung von f

f x y z = if x>42 then y+y else z^z

Normale Auswertung:

f 45 (squ (5\*(2+3))) (squ ((2+3)\*7))

(E) ->> if 45>42 then (squ (5\*(2+3))) + (squ (5\*(2+3)))  
else (squ ((2+3)\*7)) \* (squ ((2+3)\*7))

(S) ->> if True then (squ (5\*(2+3))) + (squ (5\*(2+3)))  
else (squ ((2+3)\*7))^(squ ((2+3)\*7))

(S) ->> (squ (5\*(2+3))) + (squ (5\*(2+3)))

(2E) ->> (((5\*(2+3))) \* ((5\*(2+3)))) +  
(((5\*(2+3))) \* ((5\*(2+3))))

(2S) ->> (25 \* ((5\*(2+3)))) + (((5\*(2+3))) \* ((5\*(2+3))))

(3S) ->> 625 + (((5\*(2+3))) \* ((5\*(2+3))))

(5S) ->> 625 + 625

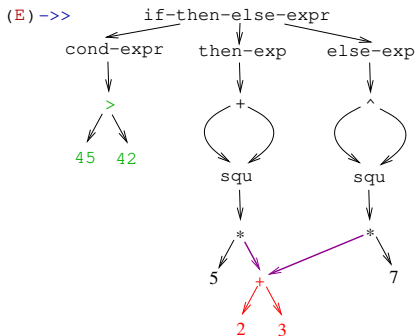
(S) ->> 1250

...das Argument (squ (5\*(2+3))) wird zweimal ausgewertet;  
das nicht benötigte Argument (squ ((2+3)\*7)) gar nicht.

# Späte, faule Auswertung von f

f x y z = if 42>x then y+y else z^z

f 45 (squ (5\*(2+3))) (squ ((2+3)\*7))



(S) ->> ... (S) ->> 1250

...das Argument (squ (5\*(2+3))) wird **genau einmal** ausgewertet; vom nicht benötigten Argument (squ ((2+3)\*7)) der Teilterm (2+3) (wg. Ausdrucksteilung ohne Extrakosten!).

# Kapitel 13.5.4

## Über Definiertheitszusammenhänge

Vortrag V

Teil V

Kap. 12

Kap. 13

13.1

13.2

13.3

13.4

13.5

13.4.1

13.5.2

13.5.3

**13.5.4**

13.5.5

13.6

13.7

13.9

Kap. 14

Umgekehrte  
Klassen-  
zim-  
mer IV

Hinweis

Aufgabe

# Definiertheitszusammenhang

...von **Argument** und **Funktion**.

**Schlüsselbegriff:** Striktheit von Funktionen.

## Definition 13.5.4.1 (Strikt im $n$ -ten Parameter)

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 (unabhängig von den Werten möglicher weiterer Argumente).

# Bsp.: 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) (LI-S) ->> undef
```

```
fac (1 'div' 0) (LÄ-E) ->>  
  if ((1 'div' 0) == 0) then 1  
    else n * fac ((1 'div' 0) - 1) (LÄ-S) ->> undef
```

```
fib (1 'div' 0) (LI-S) ->> undef
```

```
fib (1 'div' 0) (LÄ-E) ->>  
  if (1 'div' 0) == 0 then 0  
    else (1 'div' 0) == 1 then 1  
      else fib ((1 'div' 0) - 2) + fib ((1 'div' 0) - 1)  
        (LÄ-S)  
        ->> undef
```



# Bsp.: Striktheit bei mehrstelligen Funktionen

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

Der Fallunterscheidungsausdruck (-funktion)

```
(if . then . else .)
```

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

```
if ((1 'div' 0) == 0) then 4 else 2 ->> undef
```

(strikt in Bedingung)

```
if ((0 'div' 1) == 0) then 42 else 1 'div' 0  
->> if (0 == 0) then 42 else 1 'div' 0  
->> if True then 42 else 1 'div' 0  
->> 42
```

(nicht strikt im 3-ten Arg.)

```
if ((0 'div' 1) /= 0) then 1 'div' 0 else 42  
->> if (0 /= 0) then 1 'div' 0 else 42  
->> if False then 1 'div' 0 else 42  
->> 42
```

(nicht strikt im 2-ten Arg.)

## Theorem 13.5.4.2 (Striktheit, Terminierung)

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

## Korollar 13.5.4.3 (Striktheit, Ergebnisneutralität)

Durch den Übergang von später auf frühe Auswertung für strikte Argumente einer Funktion gehen keine Ergebnisse verloren (und stimmen gemäß Theorem 13.3.1 überein).

# Optimierung: Ausnutzen von Striktheit

Für **strikte Argumente** von Funktionen darf deshalb stets

- ▶ **späte** durch **frühe** Auswertung ersetzt werden

da sich Terminierungsverhalten und Resultat nicht ändern.

Da bei **früher Auswertung** Zusatzaufwand für die Verwaltung geteilter Datenstrukturen fehlt, ist die Ersetzung **später** durch **frühe** Auswertung für **strikte Funktionsargumente** eine der wichtigsten

- ▶ **Optimierungen**

bei der Übersetzung funktionaler Sprachen mit später Auswertungssemantik.

**Beispiel:** Übersetzer für Sprachen wie **Haskell** und **Miranda** dürfen also für die in ihrem jeweiligen Argument strikten **Fakultäts-** und **Fibonacci-Funktionen** **späte** durch **frühe** Argumentauswertung ersetzen.

# Optimierungsvoraussetzung: Striktheitsanalyse

Übersetzer spät auswertender Sprachen führen dazu eine sog.

- ▶ Striktheitsanalyse

durch, um dort, wo es **sicher** ist, d.h. wo ein Ausdruck zum Ergebnis beiträgt und sein Wert deshalb in **jeder** Auswertungsordnung benötigt wird,

- ▶ **späte, faule** (engl. **lazy**)

durch

- ▶ **frühe, fleißige** (engl. **eager**)

**Argumentauswertung** zu ersetzen.

Statt von **früher Auswertung** spricht man deshalb auch von

- ▶ **strikter Auswertung** (engl. **strict evaluation**).

# Kapitel 13.5.5

## Aliase applikativer, normaler Auswertung

Vortrag V

Teil V

Kap. 12

Kap. 13

13.1

13.2

13.3

13.4

13.5

13.4.1

13.5.2

13.5.3

13.5.4

**13.5.5**

13.6

13.7

13.9

Kap. 14

Umgekehrte  
Klassen-  
zim-  
mer IV

Hinweis

Aufgabe

# Aliase applikativer, normaler Auswertung

## Applikative Auswertungsordnung (engl. applicative order eval.)

- ▶ **Verwandte Bezeichnungen:** Wertparameter-, innerste, strikte Auswertung (engl. call-by-value, innermost or strict evaluation).
- ▶ **Operationalisierung:** Linksapplikative, linkestinnerste, **frühe, fleißige Auswertung** (engl. leftmost- innermost or **eager evaluation**).

## Normale Auswertungsordnung (engl. normal order evaluation)

- ▶ **Verwandte Bezeichnungen:** 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**).
  - **Verwandte Bezeichnung:** Bedarfsparameter-Auswertung (engl. call-by-need evaluation).

# Kapitel 13.6

## Frühe oder späte Auswertung? Eine Standpunktfrage

To be, or not to be,  
that is the question.

*Hamlet, Prinz von Dänemark*

William Shakespeare (um 1564-1616)  
engl. Dramatiker und Lyriker

To be *lazy*, or not to be *lazy*,  
that is the question.

...*Hamlet, zeitgerecht*  
*interpretiert*

- ▶ Frühe, fleißige Argumentauswertung (engl. *eager evaluation*), wie in *ML*, *Hope*, *Scheme* (ohne Makros),...
- ▶ Späte, faule Argumentauswertung (engl. *lazy evaluation*), wie in *Haskell*, *Miranda*, *SASL*,...

*Quot capita, tot sententiae.*

Wie viele Köpfe, so viele Ansichten.

Terenz (190 v.Chr. - 159 v.Chr.)  
röm. Schriftsteller



# Kapitel 13.6.1

## Frühe oder späte Auswertung: Vor- und Nachteile

Vortrag V

Teil V

Kap. 12

Kap. 13

13.1

13.2

13.3

13.4

13.5

13.6

**13.6.1**

13.6.2

13.7

13.9

Kap. 14

Umgekehrt

Klassen-

zim-

mer IV

Hinweis

Aufgabe

# Frühe oder späte Argumentauswertung (1)

...die Vorteile des einen sind die Nachteile des anderen und umgekehrt. Im einzelnen:

**Vorteile später Argumentauswertung** (mit Ausdrucksteilung):

- + Terminiert mit Normalform, wenn es (irgend-) eine terminierende Auswertungsreihenfolge gibt.  
**Informell:** Späte (wie normale und linksnormale) Auswertungsordnung terminieren häufigst möglich!
- + Wertet Argumente nur aus, wenn deren Werte wirklich benötigt werden; und dann nur einmal.
- + Ermöglicht eleganten und flexiblen Umgang mit potentiell unendlichen Werten von Datenstrukturen (z.B. unendliche Listen, Ströme (s. [Kap. 18.2](#)), unendliche Bäume, etc.).

# Frühe oder späte Argumentauswertung (2)

## Nachteile später Argumentauswertung:

- Konzeptuell u. implementierungstechnisch anspruchsvoller:
  - Repräsentation von Ausdrücken in Form von Graphen statt linearer Sequenzen; Ausdrucksauswertung und -manipulation als Graph- statt Sequenzmanipulation.
  - Partielle Auswertung von Ausdrücken kann Seiteneffekte bewirken! (**Beachte:** Einwand gilt nicht für Haskell; in Haskell keine Seiteneffekte! In Scheme: Seiteneffektvermeidung obliegt dem Programmierer.)
  - Ein-/Ausgabe nicht in trivialer Weise transparent für den Programmierer zu integrieren.

...volle Ursacheneinsicht erfordert tiefergehendes Verständnis von  $\lambda$ -Kalkül und Bereichstheorie (engl. **domain theory**).

# Frühe oder späte Argumentauswertung (3)

Vorteile früher Argumentauswertung:

- + Konzeptuell und implementierungstechnisch einfacher.
- + Einfache(re) Integration imperativer Konzepte.
- + Vom mathematischen Standpunkt oft 'natürlicher'.

Beispiel: Soll der Wert von Ausdrücken wie `(first (2*21, infinite))` definiert gleich `42` sein wie bei **später** Auswertung oder undefiniert wg. Nichtterminierung wie bei **früher** Auswertung?

`first (2*21, infinite)`  $\xrightarrow{\text{späte}}$  `2*21`  $\xrightarrow{\text{späte}}$  `42`

Argumentauswertung

`first (2*21, infinite)`  
 $\xrightarrow{\text{frühe}}$  `first(42, 1+infinite)`  
`frühe`  $\xrightarrow{\text{frühe}}$  `first(42, 1+(1+infinite))`  $\xrightarrow{\text{frühe}}$  ...

Arg. Auswertung

## Kapitel 13.6.2

Frühe oder späte Auswertung:  
Welche soll ich nehmen?

# Welche Auswertungsordnung soll ich nehmen?

...stets unter dem Aspekt d. Zahl der Argumentauswertungen.

## Normale Auswertungsordnung

- ▶ Argumente werden **so oft** ausgewertet, **wie** sie **verwendet** werden.
  - + Kein Argument wird ausgewertet, dessen Wert nicht benötigt wird.
  - + Terminiert, wenn immer es eine terminierende Auswertungsfolge gibt; terminiert somit am häufigsten, insbesondere häufiger als applikative Auswertung.
  - Argumente, die mehrfach verwendet werden, werden auch mehrfach ausgewertet; so oft, wie sie verwendet werden  $\rightsquigarrow$  **implementierungspraktisch deshalb irrelevant**; implementierungspraktisch relevant: **faule Auswertung**.

**Insgesamt:** Normale Auswertung ist theorie-relevant, nicht jedoch implementierungspraktisch (sehr relevant allerdings in Form **fauler Auswertung!**).

# Welche Auswertungsordnung soll ich nehmen?

## Frühe, fleißige, applikative Auswertungsordnung

- ▶ Argumente werden **genau einmal** ausgewertet.
  - + Jedes Argument wird exakt einmal ausgewertet; kein zusätzlicher Aufwand über die Auswertung hinaus.
  - Auch Argumente, deren Wert nicht benötigt wird, werden ausgewertet; das ist kritisch für Argumente, deren Auswertung teuer ist, gar nicht terminiert (unendlich teuer!) oder auf einen Laufzeitfehler führt.

## Späte, faule Auswertungsordnung (mit Ausdrucksteilung)

- ▶ Argumente werden **höchstens einmal** ausgewertet.
  - + Ein Argument wird nur ausgewertet, wenn sein Wert benötigt wird; und dann exakt einmal.
  - + Kombiniert die Vorteile von applikativer Auswertung (**Effizienz!**) und normaler Auswertung (**Terminierung!**).
  - Erfordert zusätzlichen Aufwand zur Laufzeit für die Verwaltung der Auswertung von (Teil-) Ausdrücken.

# Welche Auswertungsordnung soll ich nehmen?

...von einem pragmatischem Standpunkt aus:

- ▶ **Frühe, fleißige, applikative** Auswertung vorteilhaft gegenüber normaler und später, fauler Auswertung, da
  - + geringere Laufzeitzusatzkosten (engl. overhead).
  - + größeres Parallelisierungspotential (für Funktionsargumente).
- ▶ **Späte, faule** Auswertung vorteilhaft gegenüber früher, fleißiger, applikativer Auswertung, wenn
  - + Terminierungshäufigkeit (Definiertheit des Programms!) von überragender Bedeutung.
  - + Argumente nicht benötigt (und deshalb gar nicht ausgewertet) werden  
Bsp.:  $(\lambda x. \lambda y. y) ((\lambda x. x x)(\lambda x. x x)) z \rightarrow (\lambda y. y) z \rightarrow z$
- ▶ **Ideal: Das Beste beider Welten:**
  - + **Früh, fleißig, applikativ**, wo möglich; **spät, faul**, wo nicht.



# Zusammengefasst

...frühe, fleißige applikative Auswertung (engl. eager evaluation) oder späte, faule Auswertung (engl. lazy evaluation):

- ▶ Für beide Strategien sprechen gewichtige Gründe – und Fürsprecher:

– pro früh:

Lucundi acti labores.  
Angenehm sind die erledigten Arbeiten.

Cicero (106 - 43 v.Chr.)  
röm. Staatsmann und Schriftsteller

– pro spät:

Jetzt schau ma amoi, nacha sehn ma scho!

Franz Beckenbauer (\* 1945)  
bayer. Fußballspieler und Kaiser

- ▶ Die Wahl ist letztlich eine Frage von Angemessenheit und Zweckmäßigkeit im Anwendungskontext.

# Kapitel 13.7

## Früh(artig)e und späte Auswertung in Haskell

Vortrag V

Teil V

Kap. 12

Kap. 13

13.1

13.2

13.3

13.4

13.5

13.6

**13.7**

13.9

Kap. 14

Umgekehrt

Klassen-  
zim-  
mer IV

Hinweis

Aufgabe

# Steuerung der Auswertung in Haskell

Haskell erlaubt, die Auswertungsordnung (zu einem gewissen Grad) zu steuern.

Späte, faule Auswertung:

- ▶ Standardverfahren (vom Programmierer nichts zu tun):

```
fac (2*(3+5))
(E) ->> if (2*(3+5)) == 0 then 1
        else ((2*(3+5)) * fac ((2*(3+5))-1))
...

```

Früh(artig)e 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))
...

```

# Früh(artig)e Auswertung in Haskell (1)

Wirkung des Operators (`$!`):

- ▶ Die Auswertung des Ausdrucks (`f $! ausd`) erfolgt in gleicher Weise wie die des Ausdrucks (`f ausd`) mit dem Unterschied, dass die Auswertung von `ausd` erzwungen wird, bevor `f` angewendet und expandiert wird.

Im **Detail**: Ist der Wert von `ausd` von einem

- ▶ **elementaren Typ** wie `Int`, `Bool`, `Double`, etc., so wird `ausd` vollständig ausgewertet.
- ▶ **Tupeltyp** wie `(Int,Bool)`, `(Int,Bool,Double)`, etc., so wird `ausd` bis zu einem Tupel von Ausdrücken ausgewertet, aber nicht weiter.
- ▶ **Listentyp**, so wird `ausd` so weit ausgewertet, bis als Ausdruck die leere Liste erscheint oder die Konstruktion zweier Ausdrücke zu einer Liste.

# Früh(artig)e Auswertung in Haskell (2)

Für **curryfizierte** Funktionen kann

- **strikte** Auswertung

für **jede Argumentkombination** erreicht werden.

**Beispiel:** Für die zweistellige curryfizierte Funktion

$$f :: a \rightarrow b \rightarrow c$$

erzwingt

- $(f \$! x) y$  : Auswertung von **x**
- $(f x) \$! y$  : Auswertung von **y**
- $(f \$! x) \$! y$  : Auswertung von **x** und **y**

vor der Anwendung und **Expansion** von **f**.

# Hauptanwendung von (\$) in Haskell

...zur Minderung des Speicherverbrauchs.

Beispiel: Vergleiche Funktion

```
sumwith_lz :: Int -> [Int] -> Int
sumwith_lz v []          = v
sumwith_lz v (x:xs)     = sumwith_lz (v+x) xs
```

mit

```
sumwith_ea :: Int -> [Int] -> Int
sumwith_ea v []          = v
sumwith_ea v (x:xs)     = (sumwith_ea $! (v+x)) xs
```

# Anwendungsbsp.: Späte, faule Auswertung

...liefert:

```
sumwith_lz 36 [1,2,3]
(LÄ-E) ->> sumwith_lz (36+1) [2,3,]
(LÄ-E) ->> sumwith_lz ((36+1)+2) [3]
(LÄ-E) ->> sumwith_lz (((36+1)+2)+3) []
(LÄ-E) ->> (((36+1)+2)+3)
(LÄ-S) ->> ((37+2)+3)
(LÄ-S) ->> (39+3)
(LÄ-S) ->> 42
```

...7 Schritte; die max. Länge des summativen Terms auf erster Argumentposition hängt von der Länge der Liste auf zweiter Argumentposition ab.

# Anwendungsbsp.: Früh(artig)e Auswertung

...mittels ( $\$!$ ) liefert:

```
sumwith_ea 36 [1,2,3]
(LÄ-E) ->> (sumwith_ea $! (36+1)) [2,3]
(LI-S) ->> (sumwith_ea $! 37) [2,3]
(LI-S) ->> sumwith_ea 37 [2,3]
(LÄ-E) ->> (sumwith_ea $! (37+2)) [3]
(LI-S) ->> (sumwith_ea $! 39) [3]
(LI-S) ->> sumwith_ea 39 [3]
(LÄ-E) ->> (sumwith_ea $! (39+3)) []
(LI-S) ->> (sumwith_ea $! 42) []
(LI-S) ->> sumwith_ea 42 []
(LÄ-E) ->> 42
```

...10 Schritte, aber die Länge des summativen Terms auf erster Argumentposition ist unabhängig von der Länge der Liste auf zweiter Argumentposition und bleibt konstant kurz.



# Anwendungsbsp.: Auswertungsstile

...im Vergleich:

- ▶ Späte, faule Auswertung von `sumwith_lz 36 [1..3]`
  - baut den Ausdruck  $((36+1)+2)+3$  vollständig auf, bevor die erste Simplifikation ausgeführt wird.
  - Allgemein: `sumwith_lz` baut einen Ausdruck auf, dessen Größe proportional zur Länge der Argumentliste ist.
  - Problem: Programmabbrüche durch Speicherüberläufe schon für vergleichsweise kleine Argumente möglich:  
`sumwith_lz 36 [1..10000]`
- ▶ Früh(artig)e Auswertung von `sumwith_ea 36 [1..3]`
  - Simplifikationen werden frühestmöglich ausgeführt.
  - Exzessiver Speicherverbrauch (engl. memory leak) wird dadurch (in diesem Beispiel) vollständig vermieden.
  - Aber: Die Zahl der Rechenschritte steigt; besseres Speicherverhalten wird gegen schlechtere Schrittzahl eingetauscht (engl. trade-off).

# Zusammengefasst

Haskells ( $\$!$ )-Operator ist

- ▶ hilfreich und nützlich

das Speicherverhalten von Programmen zu verbessern, stellt allerdings keinen Königsweg dar: Auch kleine Beispiele erfordern bereits eine

- ▶ sorgfältige Untersuchung

des Verhaltens später und früh(artig)er Auswertung.

Es gibt keinen Königsweg [zur Geometrie].

Euklid (ungef. 3./4. Jhdt. v.Chr)  
griech. Mathematiker

# Kapitel 13.9

## Leseempfehlungen

Vortrag V

Teil V

Kap. 12

Kap. 13

13.1

13.2

13.3

13.4

13.5

13.6

13.7

**13.9**

Kap. 14




Umgekehrt

Klassen-  
zim-  
mer IV



Hinweis

Aufgabe





# Basisleseempfehlungen für Kapitel 13 (1)

-  Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, 2. Auflage, 1998. (Kapitel 7.1, Lazy Evaluation)
-  Richard Bird. *Thinking Functionally with Haskell*. Cambridge University Press, 2015. (Kapitel 7.1, Lazy evaluation)
-  Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2. Auflage, 2016. (Kapitel 15, Lazy evaluation; Kapitel 15.2, Evaluation strategies; Kapitel 15.7, Strict application)




# Basisleseempfehlungen für Kapitel 13 (2)

-  Greg Michaelson. *An Introduction to Functional Programming through Lambda Calculus*. Dover Publications, 2. Auflage, 2011. (Kapitel 4.4, Applicative Order Reduction; Kapitel 8, Evaluation; Kapitel 8.2, Normal Order; Kapitel 8.3, Applicative Order; Kapitel 8.8, Lazy Evaluation)
-  Simon Thompson. *Haskell – The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011. (Kapitel 17.1, Lazy evaluation; Kapitel 17.2, Calculation rules and lazy evaluation)

# Weiterführ. Leseempfehlungen für Kap. 13 (1)

-  Hendrik Pieter Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Revised Edn., North Holland, 1984. (Kapitel 13, Reduction Strategies)
-  Richard Bird, Phil Wadler. *An Introduction to Functional Programming*. Prentice Hall, 1988. (Kapitel 6.2, Models of Reduction; Kapitel 6.3, Reduction Order and Space)
-  Gilles Dowek, Jean-Jacques Lévy. *Introduction to the Theory of Programming Languages*. Springer-V., 2011. (Kapitel 2.3, Reduction Strategies)
-  Martin Erwig. *Grundlagen funktionaler Programmierung*. Oldenbourg Verlag, 1999. (Kapitel 2.1, Parameterübergabe und Auswertungsstrategien)

## Weiterführ. Leseempfehlungen für Kap. 13 (2)

-  Chris Hankin. *An Introduction to Lambda Calculi for Computer Scientists*. King's College London Publications, 2004. (Kapitel 3, Reduction; Kapitel 8.1, Reduction Machines)
-  Fethi Rabhi, Guy Lapalme. *Algorithms – A Functional Programming Approach*. Addison-Wesley, 1999. (Kapitel 3.1, Reduction Order)
-  Reinhard Wilhelm, Helmut Seidl. *Compiler Design – Virtual Machines*. Springer-V., 2010. (Kapitel 3.2, A Simple Functional Programming Language – Evaluation Strategies)

# Kapitel 14

## Typprüfung, Typinferenz

Vortrag V

Teil V

Kap. 12

Kap. 13

**Kap. 14**

14.1

14.2

14.3

14.5

14.6

14.7

Umgekehrte  
Klassen-  
zimmer  
IV

Hinweis

Aufgabe



# Kapitel 14.1

## Motivation

Vortrag V

Teil V

Kap. 12

Kap. 13

Kap. 14

**14.1**

14.2

14.3

14.5

14.6

14.7

Umgekehrte  
Klassen-  
zim-  
mer IV

Hinweis

Aufgabe

# Typisierte Programmiersprachen

...teilen sich in

- ▶ **schwach** (Typprüfung zur **Laufzeit**)
- ▶ **stark** (Typprüfung/-inferenz zur **Übersetzungszeit**)

getypte Sprachen.

Typfehler werden deshalb in

- ▶ **schwach** getypten Sprachen **erst zur Laufzeit**
- ▶ **stark** getypten Sprachen **bereits zur Übersetzungszeit**

erkannt.

**Haskell** ist eine **stark getypte Programmiersprache!**

# Vorteile stark typisierter Programmiersprachen

- + **Verlässlicherer Code:** Der Nachweis der **Typkorrektheit** ist ein **Korrektheitsbeweis** für ein Programm auf dem **Abstraktionsniveau von Typen**. Viele Programmier- und Tippfehler werden dadurch schon zur Übersetzungszeit entdeckt.
- + **Effizienterer Code:** Keine Typprüfungen zur Laufzeit nötig.
- + **Effektivere Programmentwicklung:** Typinformation ist **Programmdokumentation** und vereinfacht **Verstehen**, **Wartung** und **Weiterentwicklung** eines Programms, z.B. auch bei der Suche nach vordefinierten Bibliotheksfunktionen: **“Gibt es eine Funktion, die alle Duplikate aus einer Liste entfernt?”**  
In Haskell kann so die Suche eingeschränkt werden auf Funktionen mit Typ  $(Eq\ a \Rightarrow [a] \rightarrow [a])$ .

# Für Haskell gilt

- ▶ Gültige Ausdrücke haben wohldefinierte Typen und heißen wohlgetypt.
- ▶ Typen wohlgetypter Ausdrücke können sein:
  - Monomorph  
`fac :: Int -> Int`  
Erkennungszeichen: Keine Typvariablen, nur konkrete Typen in der Signatur.
  - Parametrisch polymorph (uneingeschränkt polymorph)  
`length :: [a] -> Int`  
Erkennungszeichen: Typvariablen, keine Typkontexte.
  - *Ad hoc* polymorph (eingeschränkt polymorph)  
`elem :: Eq a => a -> [a] -> Bool`  
Erkennungszeichen: Typvariablen und Typkontexte.
- ▶ Typen können angegeben sein:
  - explizit: Typprüfung (grundsätzlich) ausreichend.
  - implizit: Typinferenz erforderlich.

# Typprüfung, Typinferenz

...sind **Schlüsselfähigkeiten** von Übersetzern, Interpretierern.

Der Typ des Ausdrucks:

```
magicType = let
    pair x y z = z x y
    f y = pair y y
    g y = f (f y)
    h y = g (g y)
in h (\x -> x)
```

...kann **automatisch inferiert** werden und zeigt, wie mächtig **automatische Typinferenz** ist.

# Automatische Typinferenz mit Hugs

...für `magicType`.

Das Hugs-Kommando `:type` (oder kürzer `:t`):

```
Main>:t magicType
```

liefert:

```
magicType ::
```

```
(((((a -> a) -> (a -> a) -> b) -> b) ->
((a -> a) -> (a -> a) -> b) -> b) -> c) -> c) ->
(((a -> a) -> (a -> a) -> b) -> b) ->
(((a -> a) -> (a -> a) -> b) -> b) -> c) -> c) -> d) -> d) ->
((((a -> a) -> (a -> a) -> b) -> b) -> ((a -> a) ->
(a -> a) -> b) -> b) -> c) -> c) -> (((a -> a) ->
(a -> a) -> b) -> b) -> ((a -> a) ->
(a -> a) -> b) -> b) -> c) -> c) -> d) -> d) -> e) -> e
```

# Wie gelingt es

...Übersetzern, Interpretierern, **komplexe Typen** wie den von **magicType** **automatisch zu inferieren?**

**Informell:** Durch Auswertung jeder Art von

- **Kontextinformation** in Ausdrücken, Funktionsdefinitionen und Typklassen, z.B. verwendete Operatoren ((+), (/), (&&), (++)), Konstanten (2, 3.57, True, (2,3), [], [2,3,4],...), Muster (('c':cs), (x,False),...), etc.

Anhand von Beispielen betrachten wir **Methoden** und **Vorgehensweisen** für:

- **Typanalyse, Typprüfung**
- **Unifikation** (s. vollst. LVA-Unterlagen)
- **Typsysteme, Typinferenz**

# Kapitel 14.2

## Monomorphe Typprüfung

Vortrag V

Teil V

Kap. 12

Kap. 13

Kap. 14

14.1

**14.2**

14.3

14.5

14.6

14.7

Umgekehrte  
Klassen-  
zimmer  
IV

Hinweis

Aufgabe



# Am Ende monomorpher Typprüfung

...steht als Ergebnis:

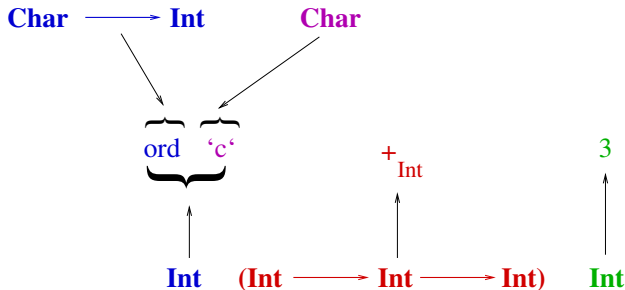
Ein **Ausdruck** ist

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

# Typprüfung für Ausdrücke, monomorpher Fall

Bsp. 1: Ausdruck  $(\text{ord } 'c' +_{\text{Int}} 3)$

Durch **Kontextauswertung** des Ausdrucks kann **Typprüfung**:

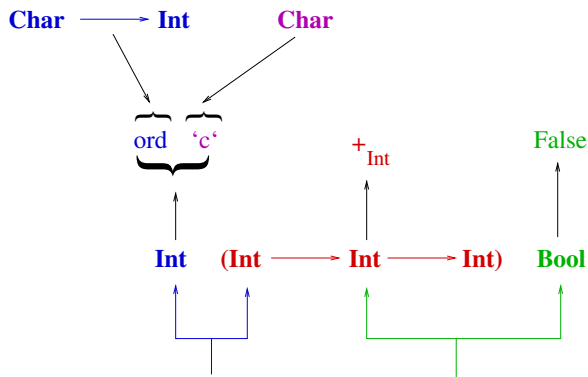


...korrekte Typung nachweisen!

# Typprüfung für Ausdrücke, monomorpher Fall

Bsp. 2: Ausdruck (ord 'c' +<sub>Int</sub> False)

Durch Kontextauswertung des Ausdrucks kann Typprüfung:



Erwarteter und tatsächlicher Typ, **Int**,  
stimmen überein: **Typkorrekt!**

Erwarteter Typ, **Int**, und tatsächlicher  
Typ, **Bool**, stimmen nicht überein: **Typinkorrekt!**

...inkorrekte Typung aufdecken!

# Typprüfung für Fkt.-Def., monomorpher Fall

Bsp. 3:  $f$  monomorphe Fkt.-Def., d.h.  $t_i$ ,  $t$  konkrete Typen:

```
f :: t1 -> t2 -> ... -> tk -> t
f m1 m2 ... mk
| w1 = a1
| w2 = a2
...
| wn = an
```

...für die Typprüfung von  $f$  sind 3 Kontexteigenschaften auszunutzen:

1. Jeder **Wächter**  $w_i$  muss vom Typ **Bool** sein.
2. Jeder **Ausdruck**  $a_i$  muss vom Resultattyp  $t$  sein.
3. Das **Muster** jedes Parameters  $m_i$  muss mit dem zugehörigen Parametertyp  $t_i$  **konsistent** sein.

# Kapitel 14.3

## Polymorphe Typprüfung

Vortrag V

Teil V

Kap. 12

Kap. 13

Kap. 14

14.1

14.2

**14.3**

14.5

14.6

14.7

Umgekehrte  
Klassen-  
zim-  
mer IV

Hinweis

Aufgabe

# Am Ende polymorpher Typprüfung

...steht als Ergebnis:

Ein **Ausdruck** ist

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

**Algorithmischer Schlüssel** zu **polymorpher Typprüfung** ist das

- **Lösen** von **Typkontextsystemen** (engl. **constraint satisfaction**)

unter **Unifikation** von **Typausdrücken**.

# Typprüfung, polymorpher Fall (1)

Bsp. 1: Die polymorphe Funktionssignatur :

```
length :: [a] -> Int
```

...informell steht der Typausdruck  $([a] \rightarrow \text{Int})$  für die unendliche Menge konkreter Typen  $([\tau] \rightarrow \text{Int})$  mit  $\tau$  beliebiger monomorpher Typ, z.B.:

```
[Int] -> Int
```

```
[(Bool,Char)] -> Int
```

```
[String -> String] -> Int
```

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

```
...
```

# Typprüfung, polymorpher Fall (2)

...in Aufrufkontexten wie:

1. `length [length [1,2,3], length [True,False,True], length [], length [(+),(*),(-)]]`
2. `length [(True,'a'), (False,'q'), (True,'o')]`
3. `length [reverse, ("Felix" ++), tail, init]`
4. `length [(&&), (||), xor, nand, nor]`

...lässt sich ein monomorphe Typ eindeutig erschließen:

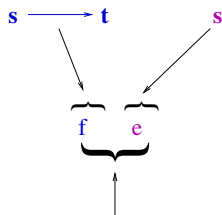
1. `length :: [Int] -> Int`
2. `length :: [(Bool,Char)] -> Int`
3. `length :: [(String -> String)] -> Int`
4. `length :: [(Bool -> Bool -> Bool)] -> Int`

**Beachte:** Der Kontext `length []` erlaubt nur auf `[a] -> Int` für den Typ zu schließen (Übungsaufgabe: Warum?).



# Typprüfung, polymorpher Fall (3)

Bsp. 2: Der applikative Ausdruck  $(f\ e)$ :



$(f\ e)$  hat (Resultat-) Typ  $t$

Ist weitere Kontextinformation für  $f$  und  $e$  nicht vorhanden, liefert die **Kontextauswertung** von  $(f\ e)$  für die **allgemeinst möglichen** Typen von  $e$ ,  $f$  und  $(f\ e)$ :

1.  $f$  wird auf ein Argument appliziert;  $f$  muss deshalb von fkt. Typ sein:  $f :: s \rightarrow t$
2. Der Typ von Ausdruck  $e$  muss deshalb sein:  $e :: s$
3. Und folglich der Typ von Ausdruck  $f\ e$ :  $f\ e :: t$

# Typprüfung, polymorpher Fall (4)

Bsp. 3: Die Funktionsgleichung:

$$f(x, y) = (x, ['a' .. y])$$

Die **Kontextauswertung** liefert:  $f$  bezeichnet eine Funktion, die als Argument **Paare** erwartet, an deren

- **1-te Komponente** keine Bedingung gestellt ist, die also von einem beliebigen Typ sein darf.
- **2-te Komponente** eine Bedingung gestellt ist:  $y$  muss vom Typ **Char** sein, da  $y$  als Schranke des Zeichenreihenwerts  $['a' .. y]$  benutzt wird.

Beides zusammen erlaubt den **allgemeinsten Typ** von  $f$  zu erschließen:

$$f :: (a, Char) \rightarrow (a, [Char])$$

# Typprüfung, polymorpher Fall (5)

Bsp. 4: Die Funktionsgleichung:

$$g(m, zs) = m + \text{length } zs$$

Die **Kontextauswertung** ergibt:  $g$  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  $\text{length}$  verwendet wird, die den Typ  $([b] \rightarrow \text{Int})$  hat.

Beides zusammen erlaubt den **allgemeinsten Typ** von  $g$  zu erschließen:

$$g :: (\text{Int}, [b]) \rightarrow \text{Int}$$

# Typprüfung für Fkt.-Def., polymorpher Fall

Bsp. 6:  $f$  echt polymorphe Fkt.-Def., d.h.  $b_i$ ,  $b$  Typvariablen:

$f :: b_1 \rightarrow b_2 \rightarrow \dots \rightarrow b_k \rightarrow b$

$f \ m_1 \ m_2 \ \dots \ m_k$

|  $w_1 = a_1$

|  $w_2 = a_2$

...

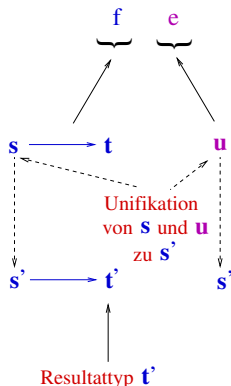
|  $w_n = a_n$

...für die Typprüfung von  $f$  sind 3 Kontexteigenschaften auszunutzen:

1. Jeder **Wächter**  $w_i$  muss vom Typ **Bool** sein.
2. Jeder **Ausdruck**  $a_i$  muss von einem Typ  $t_i$  sein, der mindestens (! – umgekehrt im Aufruffall) so allgemein ist wie der Typ  $b$ , d.h.  $b$  muss eine Instanz von  $t_i$  sein.
3. Das **Muster** jedes Parameters  $m_i$  muss mit dem zugehörigen Parametertyp  $b_i$  konsistent sein.

# Unifikation bei Typprüfung v. Fkt.-Termen

Bsp. 7: Der Funktionsterm  $(f\ e)$ :



Es gilt: Typkorrektheit von  $(f\ e)$  erfordert nicht Gleichheit von  $s$  und  $u$ ; es reicht, wenn sie unifizierbar sind: Unifizierter Typ von  $f$  ist  $(s' \rightarrow t')$ , von  $(f\ e)$  somit  $t'$ .

# Zusammenfassend: Typprüfung, polym. Fall

...die Beispiele zeigen, dass wie im **monomorphen** Fall die **Anwendungskontexte** von Ausdrücken und Funktionsdefinitionen implizit ein

- System von **Typbedingungen** festlegen.

Das **Typprüfungsproblem** reduziert sich so auf die Bestimmung der

- **allgemeinst möglichen** Typausdrücke, so dass **keine** Bedingung verletzt ist.

Dafür ist i.a. die **Unifikation von Typausdrücken** nötig (siehe vollst. LVA-Unterlagen).

# Kapitel 14.5

## Typsysteme, Typinferenz

Vortrag V

Teil V

Kap. 12

Kap. 13

Kap. 14

14.1

14.2

14.3

**14.5**

14.6

14.7

Umgekehrte  
Klassen-  
zim-  
mer IV

Hinweis

Aufgabe

# Typsysteme, Typinferenz

## 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.

## Typinferenz bezeichnet

- den Prozess, den Typ eines Ausdrucks automatisch mithilfe der Axiome und Regeln des Typsystems abzuleiten.



# Typgrammatik (typischer Ausschnitt)

...erzeugt eine **Typsprache**:

$$\begin{array}{l} \tau ::= \textit{Int} \mid \textit{Float} \mid \textit{Char} \mid \textit{Bool} \\ \quad \mid \alpha \\ \quad \mid \tau \rightarrow \tau \end{array} \quad \begin{array}{l} \text{(Einfacher Typ)} \\ \text{(Typvariable)} \\ \text{(Funktionstyp)} \end{array}$$
$$\begin{array}{l} \sigma ::= \tau \\ \quad \mid \forall \alpha. \sigma \end{array} \quad \begin{array}{l} \text{(Typ)} \\ \text{(Typbindung)} \end{array}$$

Sprechweisen:  $\tau$  ist ein **Typ**,  $\sigma$  ein **Typschema**.

# Typsystem (typischer Ausschnitt)

...assoziiert mit jedem (typisierbaren) Ausdruck der Sprache einen **Typ** der Typsprache, wobei  $\Gamma$  eine sogenannte **Typannahme** (oder **Typumgebung**) ist:

Axiome:

$$\text{VAR} \quad \frac{\text{---}}{\Gamma \vdash \text{var} : \Gamma(\text{var})}$$

$$\text{CON} \quad \frac{\text{---}}{\Gamma \vdash \text{con} : \Gamma(\text{con})}$$

$$\text{COND} \quad \frac{\Gamma \vdash \text{exp} : \text{Bool} \quad \Gamma \vdash \text{exp}_1 : \tau \quad \Gamma \vdash \text{exp}_2 : \tau}{\Gamma \vdash \text{if exp then exp}_1 \text{ else exp}_2 : \tau}$$

Regeln:

$$\text{APP} \quad \frac{\Gamma \vdash \text{exp} : \tau' \rightarrow \tau \quad \Gamma \vdash \text{exp}' : \tau'}{\Gamma \vdash \text{exp exp}' : \tau}$$

$$\text{ABS} \quad \frac{\Gamma[\text{var} \mapsto \tau'] \vdash \text{exp} : \tau}{\Gamma \vdash \lambda x. \text{exp} : \tau' \rightarrow \tau}$$

...

# Pragmatisch wichtig

...ist es Typsysteme so anzugeben, dass stets nur

- ein Axiom oder eine Regel anwendbar ist.

Man spricht dann von **syntaxisgerichteter** Anwendbarkeit der Inferenzregeln, was zu

- effizienteren Typinferenzalgorithmen führt.

# Kapitel 14.6

## Zusammenfassung

Vortrag V

Teil V

Kap. 12

Kap. 13

Kap. 14

14.1

14.2

14.3

14.5

**14.6**

14.7

Umgekehrte  
Klassen-  
zimmer  
IV

Hinweis

Aufgabe

# Zusammenfassung

Haskell ist stark typisiert:

- Wohltypisierung von Haskell-Programmen ist deshalb zur Übersetzungszeit entscheidbar. Fehler zur Laufzeit aufgrund von Typfehlern sind deshalb ausgeschlossen.
- Typen können, müssen aber vom Programmierer nicht angegeben werden.
- Übersetzer und Interpretierer inferieren die Typen von Ausdrücken und Funktionsdefinitionen (in jedem Fall automatisch).

Lohnt sich dann die Mühe, Typen explizit zu spezifizieren?

Ja, im Programm angegebene Typen sind nützlich für

- Programmierer als Programmkommentierung.
- Interpretierer und Übersetzer als Hilfe, aussagekräftigere Fehlermeldungen zu erzeugen.

# Kapitel 14.7

## Leseempfehlungen

Vortrag V

Teil V

Kap. 12

Kap. 13

Kap. 14

14.1

14.2

14.3

14.5

14.6





**14.7**

Umgekehrte  
Klassen-  
zim-  
mer IV





Hinweis

Aufgabe

# Basisleseempfehlungen für Kapitel 14

-  Antonie J.T. Davie. *An Introduction to Functional Programming Systems using Haskell*. Cambridge University Press, 1992. (Kapitel 4.7, Type Inference)
-  Martin Erwig. *Grundlagen funktionaler Programmierung*. Oldenbourg Verlag, 1999. (Kapitel 5, Typisierung und Typinferenz)
-  Anthony J. Field, Peter G. Robinson. *Functional Programming*. Addison-Wesley, 1988. (Kapitel 7, Type inference systems and type checking)
-  Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011. (Kapitel 13, Overloading, type classes and type checking)

# Weiterführende Leseempfehlungen für Kap. 14

-  Luca Cardelli. *Basic Polymorphic Type Checking*. Science of Computer Programming 8:147-172, 1987.
-  Luís Damas, Robin Milner. *Principal Type Schemes for Functional Programming Languages*. In Conference Record of the 9th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'82), 207-218, 1982.
-  Gilles Dowek, Jean-Jacques Lévy. *Introduction to the Theory of Programming Languages*. Springer-V., 2011. (Kapitel 6, Type Inference; Kapitel 6.1, Inferring Monomorphic Types; Kapitel 6.2, Polymorphism)
-  John C. Mitchell. *Type Systems for Programming Languages*. In *Handbook of Theoretical Computer Science, Vol. B: Formal Methods and Semantics*, Jan van Leeuwen (Hrsg.). Elsevier Science Publishers, 367-458, 1990.



# Umgekehrtes Klassenzimmer IV

*...zur Übung, Vertiefung*

...nach Eigenstudium von Teil IV 'Fkt. Programmierung':

- Zwar weiß ich viel...

Als Bonusthema, so weit die Zeit erlaubt:

- The (Hi)Story of Haskell

Zwar weiß ich viel...

doch möchte ich alles wissen.

*Wagner, Assistent von Faust*  
Johann Wolfgang von Goethe (1749-1832)  
dt. Dichter und Naturforscher

Vortrag V

Teil V

Kap. 12

Kap. 13

Kap. 14

Umgekehr

Klassen-

zim-

mer IV

Zwar weiß  
ich viel...

Bonusthema:  
The  
(Hi)Story of  
Haskell

Hinweis

Aufgabe

# Zeit für Ihren Zweifel, Ihre Fragen!

Der Zweifel ist der Beginn der Wissenschaft.

Wer nichts anzweifelt, prüft nichts.

Wer nichts prüft, entdeckt nichts.

Wer nichts entdeckt, ist blind und bleibt blind.

Pierre Teilhard de Chardin (1881-1955)

franz. Jesuit, Theologe, Geologe und Paläontologe

Die großen Fortschritte in der Wissenschaft  
beruhen oft, vielleicht stets, darauf, dass man  
eine zuvor nicht gestellte Frage doch,  
und zwar mit Erfolg, stellt.

Carl Friedrich von Weizsäcker (1912-2007)

dt. Physiker und Philosoph

...entdecken Sie den **Wagner** in sich!

# Bonusthema

## The (Hi)Story of Haskell

...erzählt von [Simon Peyton Jones](#) in einem eingeladenen Hauptvortrag auf der 'POPL 2003'-Konferenz:

[Simon Peyton Jones. Wearing the Hair Shirt: A Retrospective on Haskell.](#)  
<https://www.microsoft.com/en-us/research/publication/wearing-hair-shirt-retrospective-haskell-2003/>

“Haskell was 15 years old at the POPL'03 meeting, when I presented this talk: it was born at a meeting at the 1987 conference on Functional Programming and Computer Architecture (FPCA'87).

In this talk, which is very much a personal view, I take a look back at the language, and try to tease out what we have learned from the experience of designing and implementing it. The main areas I discuss are: syntax (briefly), laziness (the hair shirt of the title), type classes, and sexy types.

On the way, I try to identify a few open questions that I think merit further study.”

# „Ähnlich, aber zeitlich später:

Simon Peyton Jones. (Failing to) avoid Success at all Costs: the Haskell Story.

Veranstalter: Cambridge University Computing and Technology Society, <https://cucats.org/event/12>

Vortragsfolien: <https://cucats.org/files/Escape%20from%20the%20ivory%20tower%20Feb12.pdf>

“Haskell is twenty one years old, an age at which most programming languages are either dead and buried, or else have become mainstream and hence frozen in a web of backward-compatibility constraints. Haskell is different: it is in rude health, is widely used (but not too widely!), and is still in a state of furious innovation.

In this talk I'll reflect on this two-decade journey, I'll discuss Haskell's birth and evolution, including some of the research and engineering challenges we faced in design and implementation. I'll focus particularly on the ideas that have turned out, in retrospect, to be most important and influential, as well as sketching some current developments and making some wild guesses about the future.”

# Auch als Vortragsaufzeichnung

...hier z.B. aus dem Jahr 2017 am Churchill College, University of Cambridge, UK:

[Simon Peyton Jones. Escape from the Ivory Tower: The Haskell Journey, 1:04:16.](#)

<https://www.chu.cam.ac.uk/news/2017/may/9/annual-computer-science-lecture-2017/>

In this talk Simon discusses Haskell's birth and evolution, including some of the research and engineering challenges faced in its design and implementation. Focusing particularly on the ideas that have turned out, in retrospect, to be most important and influential, as well as sketching some current developments and making some wild guesses about the future.

# Hinweis

...für das Verständnis von **Vorlesungsteil V** ist eine über den unmittelbaren Inhalt von **Vortrag V** hinausgehende weitergehende und vertiefende Beschäftigung mit dem Stoff nötig; siehe:

- ▶ **vollständige Lehrveranstaltungsunterlagen**

...verfügbar auf der Webseite der Lehrveranstaltung:

[http://www.complang.tuwien.ac.at/knoop/fp185A05\\_ws2021.html](http://www.complang.tuwien.ac.at/knoop/fp185A05_ws2021.html)

# Aufgabe bis Mittwoch, 02.12.2020

...selbstständiges Durcharbeiten von Teil V 'Funktionale Programmierung', Kap. 12, 13 und 14 und von Leit- und Kontrollfragenteil V zur Selbsteinschätzung und als Grundlage für die umgekehrte Klassenzimmersitzung am 02.12.2020:

Vortrag, umgek. Klassenz.	Thema Vortrag	Thema umgek. Klassenz.
Di, 06.10.2020, 08:15-09:45	Teil I	n.a. / Vorbesprechung
Di, 13.10.2020, 08:15-09:45	Teil II	Teil I
Di, 27.10.2020, 08:15-09:45	Teil III	Teil II
Mi, 04.11.2020, 08:15-09:45	Teil IV	Teil III
Mi, 18.11.2020, 08:15-09:45	<b>Teil V</b>	<b>Teil IV</b>
Mi, 02.12.2020, 08:15-09:45	<b>Teil VI</b>	<b>Teil V</b>
Mi, 16.12.2020, 08:15-09:45	Teil VII	Teil VI

Vortrag V

Teil V

Kap. 12

Kap. 13

Kap. 14

Umgekehrte  
Klassen-  
zim-  
mer IV

Hinweis

Aufgabe