Typsystemen

Contents

[Grundlagen 2](#_Toc68271392)

[Regeln (Bedeutung) 2](#_Toc68271393)

[Const 2](#_Toc68271394)

[Var 2](#_Toc68271395)

[Abstraktion 2](#_Toc68271396)

[Applikation 2](#_Toc68271397)

[Regeln (Abbildung) 3](#_Toc68271398)

[Typherleitung 3](#_Toc68271399)

[Typisierbare Lambda-Terme 3](#_Toc68271400)

[Polymorhpie 4](#_Toc68271401)

[Typschemata 4](#_Toc68271402)

[Angepasste Regeln 4](#_Toc68271403)

[Let-Polymorphismus 4](#_Toc68271404)

[Herleitungsbaum für ein Lambda-Ausdruck erstellen 5](#_Toc68271405)

[Ausdruck 5](#_Toc68271406)

[Baum 5](#_Toc68271407)

[Schritte 5](#_Toc68271408)

[Constraints für Unifikation (C): 6](#_Toc68271409)

[mgu bestimmen (σ\_c) 6](#_Toc68271410)

[Lambda Ausdrücke Typisieren 6](#_Toc68271411)

[Identität 6](#_Toc68271412)

[Y-Kombinator 7](#_Toc68271413)

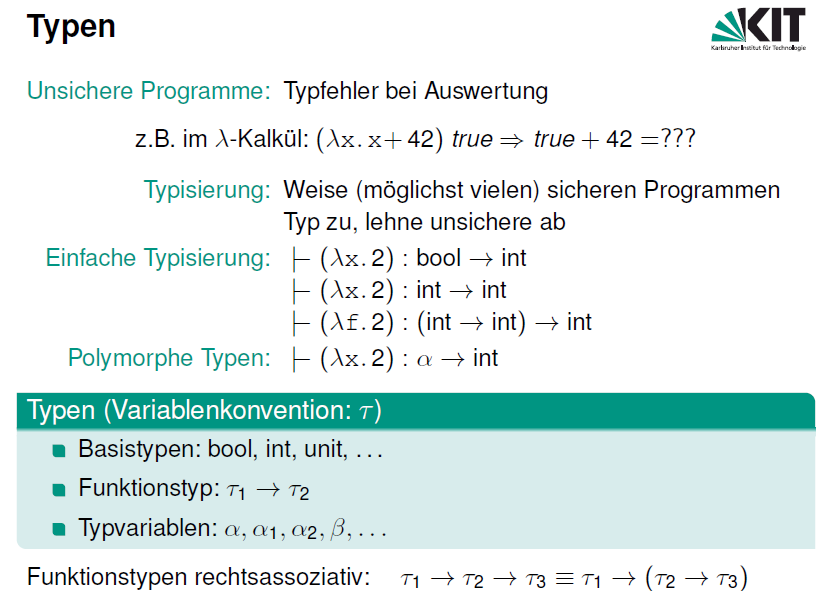
[Weitere nicht Typisierbate Terme 7](#_Toc68271414)

[FAQ 7](#_Toc68271415)

[Was macht Г? 7](#_Toc68271416)

# Grundlagen

**Funktionstypen sind rechtsassoziativ**

****

## Regeln (Bedeutung)

Const: Jede Konstante hat i.A ein eigenen Typ: 42 : t\_42, 43 : t\_43 usw. Das ersparen wir uns und sagen, dass jede Konstante einen Typ t\_c hat. -> Für jede Konstante ist Typ fest.

z.B 42: int

Var: variable x hat den Typ t, wenn in Г vermerkt ist, dass die Variable x den Typ t hat

Abstraktion: (== Lambda-Abstraktion) Wir wollen den Typ des Lambda Ausdrucks bestimmen (Funktionstyp, wie t1 -> t2). Funktionsrumpf t hat Typ t2. t2 ist dann Typ des Funktionswertes (Ausgabe der Funktion). Dann ist t2 auch das Ergebnistyp der Lambda.

Normalerweise kommt x in t vor. (x ist frei in t).

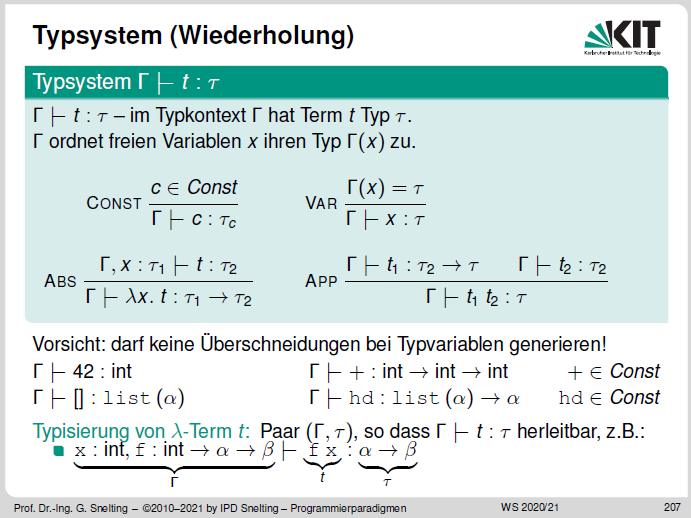
In Г soll man merken, dass x der Typ t1 hat.

Dann gilt \x . t : t1 -> t2

Also wenn gilt \x . t : t1 -> t2, dann kann man annehmen, dass x den Typ t1 hat -> nach Г schreiben.

Applikation: Funktion t1 wird angewendet auf den aktuellen Parameter t2, und das Ding wollen wir typisieren. t1 muss ein Funktion sein

## Regeln (Abbildung)



## Typherleitung

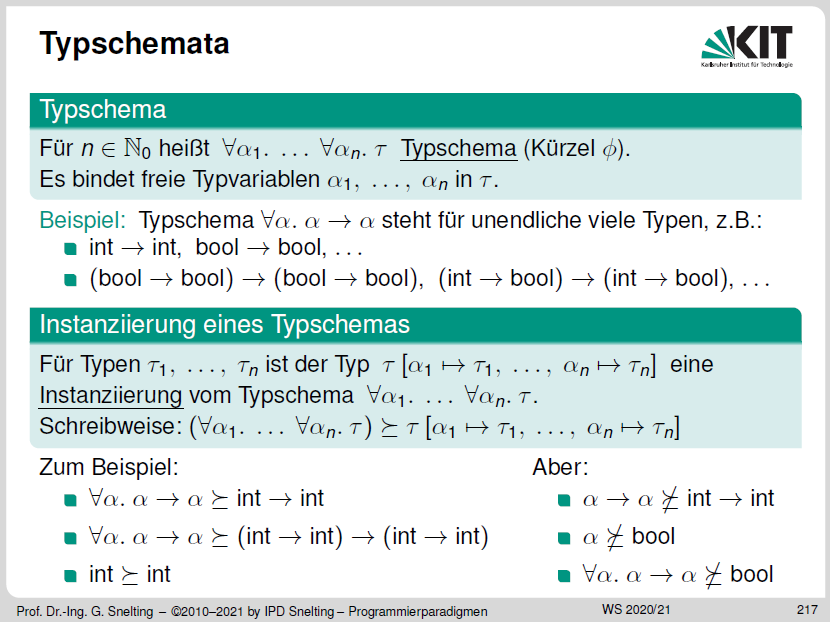
Man versucht Baum rückwärts zu konstruieren. Man fängt bei Zielaussage an und versucht die Voraussetzungen zu erfüllen.

## Typisierbare Lambda-Terme

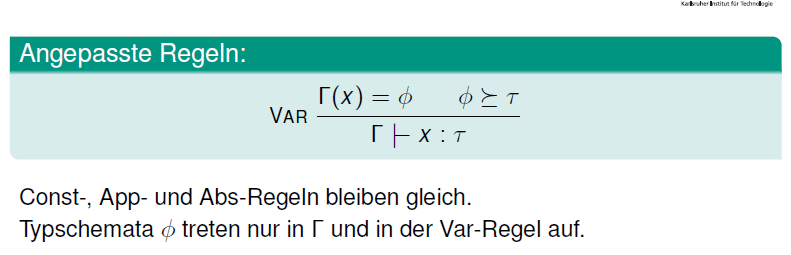
t ist typisierbar in Context Г, wenn s mit Г :- t : s existiert

# Polymorhpie

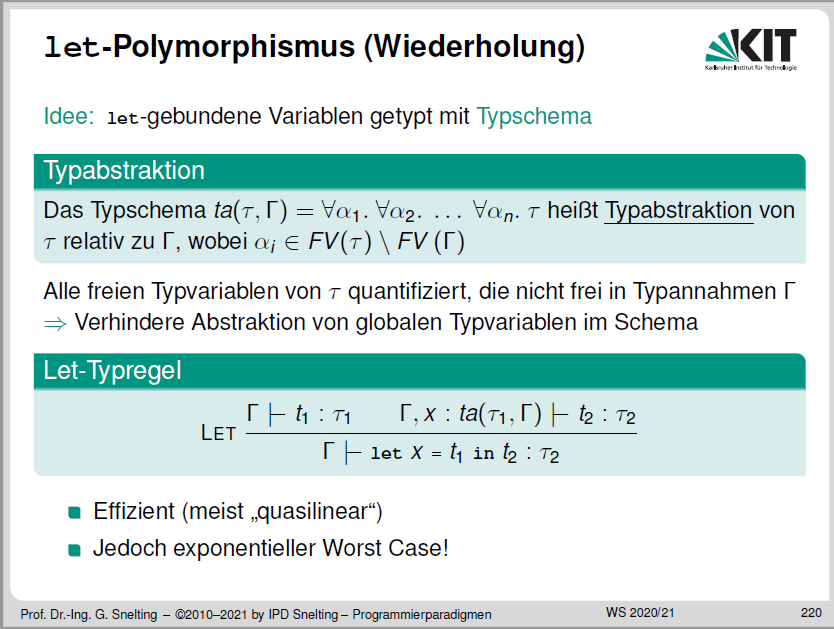
## Typschemata



## Angepasste Regeln



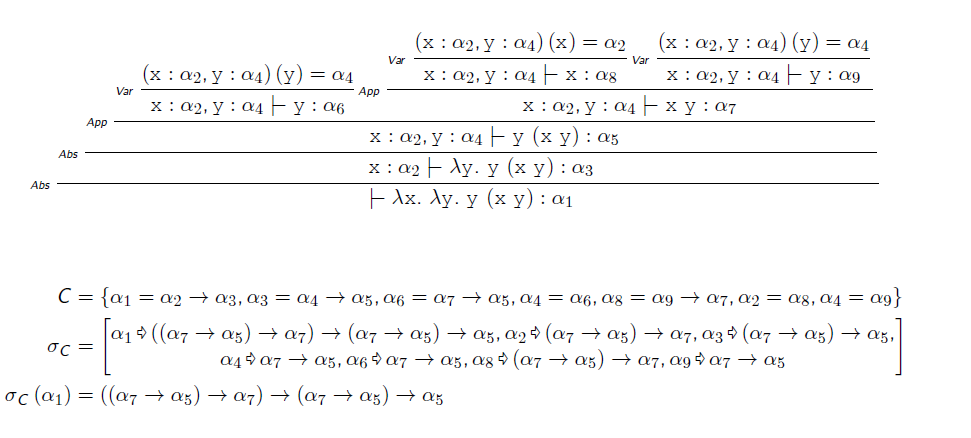
## Let-Polymorphismus



## Herleitungsbaum für ein Lambda-Ausdruck erstellen

Ausdruck: **\x. \y. y (x y)**

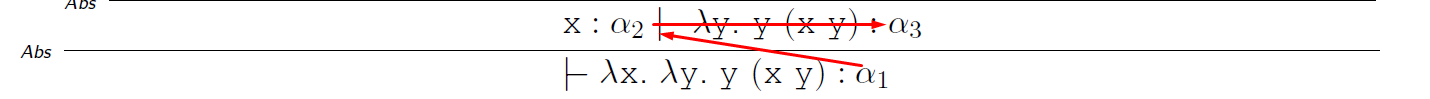
### Baum

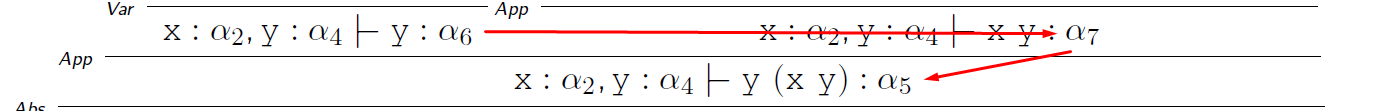


### Schritte

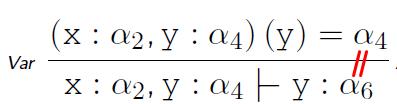
1. Linke Lambda -> Abs-Regel
2. Falls es keine linke Lambda gibt, aber man der rechte Ausdruck auswerten kann -> App-Regel
3. Wenn es nur eine Variable/Konstante steht, dann Var-/Cons- Regel

### Constraints für Unifikation (C):

Abs: a1 = a2 --> a3

App: a6 = a7 --> a5, es kommt weiter keine Regel für a5 

Var/Char: a4 = a6



### mgu bestimmen (σ\_c)

1. Schreibe alle Variablen in einer Spalte (a1, a2, a3...)t, mache viel Platz zwischen Zeilen
2. Versuche alle Variable von hinten nach vorne zu „öffnen“, bis es keine weitere Substituion möglich ist :
   1. Original: a4 = a5 -> a6, a5 = a7 -> a8
   2. Subtitutiiert: a4 = (a7 -> a8) -> a6
   3. Regel: a4 => (a7 -> a8) -> a6
   4. **Immer klammern!**
3. Suche nach “implizite Gleichheiten”:
   1. Gegeben:
      1. a8 = a2 = a6
      2. a6 = a7 -> a5
      3. a8 = a9 -> a7
   2. Da a8 = a6, ist auch a7=a9 und a5=a7
   3. Neue Regeln: a7 -> a9, a5 -> a9
4. Benutze neue Wissen, um die Regeln aus 3) noch zu unifizieren
5. Öffne a1 am Ende
6. Prüfe alles noch mal

# Lambda Ausdrücke Typisieren

## Identität

(\x. x) ist beliebig typisierbar:

int -> int

[a] -> [a]

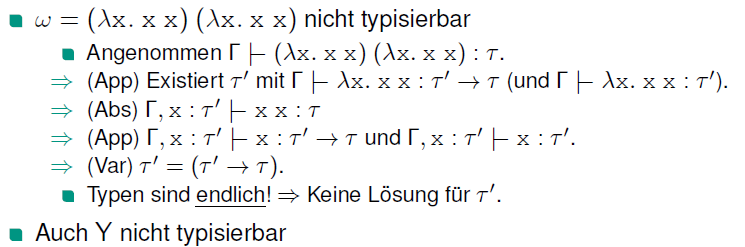
a -> a

(a -> a) -> (a -> a)

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

## Y-Kombinator

Nicht Typisierbar



## Weitere nicht typisierbate Terme

Text

Description automatically generated

# FAQ

## **Was macht Г?**

Г : Tabelle, wo für jede frei Variable typ t steht