Typsystemen

Contents

[Grundlagen 1](#_Toc67221503)

[Regeln: 1](#_Toc67221504)

[Г 1](#_Toc67221505)

[Const 2](#_Toc67221506)

[Var 2](#_Toc67221507)

[Abstraktion 2](#_Toc67221508)

[Applikation 2](#_Toc67221509)

[Abbildung 2](#_Toc67221510)

[Typherleitung: 3](#_Toc67221511)

[Typisierbare Lambda-Terme: 3](#_Toc67221512)

[Polymorhpie 3](#_Toc67221513)

[Typschemata 3](#_Toc67221514)

[Regeln 4](#_Toc67221515)

[4](#_Toc67221516)

[Let-Polymorphismus: 4](#_Toc67221517)

# Grundlagen

## Regeln:

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

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).

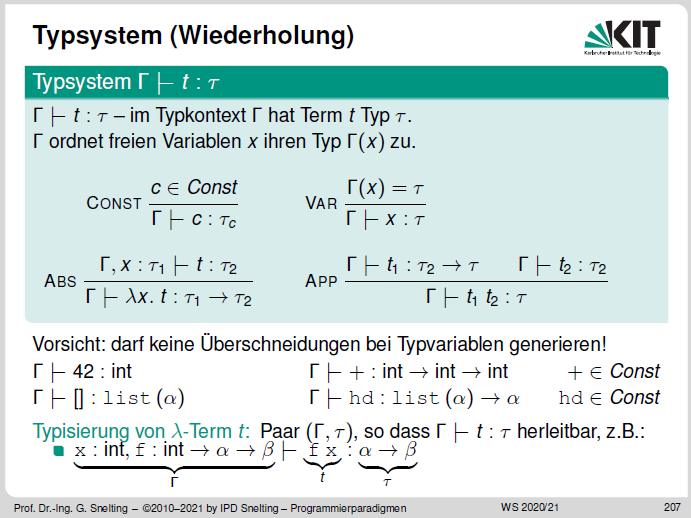
In Г soll man merken, dass x der Typ t1 hat.

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

## Abbildung



## Typherleitung:

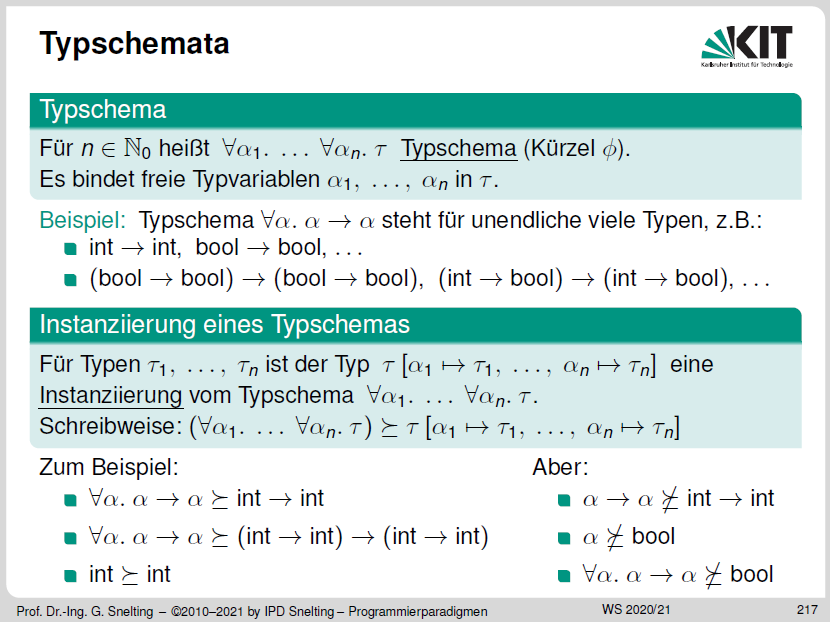
Man versucht Baum rückwärts zu konstruieren. Man fängt bei Zielaussage an und versucht die Voraussetzungen zu erfüllen.

## Typisierbare Lambda-Terme:

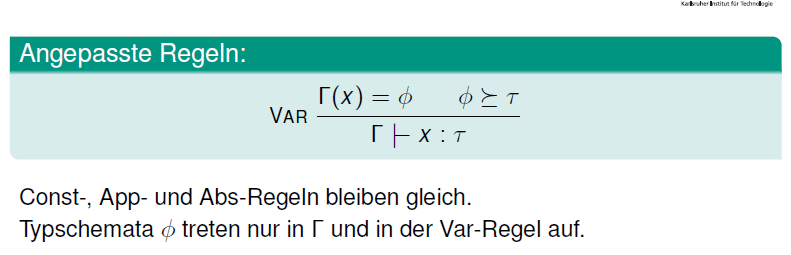
t ist typisierbar in Context Г, wenn s mit Г :- t : s existiert

# Polymorhpie

## Typschemata



## Regeln



## Let-Polymorphismus:

