

# Inhaltsverzeichnis

<b>1</b>	<b>Theoretische Grundlagen</b>	<b>2</b>
1.1	Äquivalenz . . . . .	2
1.1.1	$\alpha$ -Äquivalenz (Folie: 20_(166)) . . . . .	2
1.1.2	$\eta$ -Äquivalenz (Folie: 20_(167)) . . . . .	2
1.1.3	Church-Zahlen (Folie: 20_(179ff)) . . . . .	2
1.1.4	Rekursionsoperator Y (Folie: 20_(184)) . . . . .	3
1.2	Typinferenz . . . . .	3
<b>2</b>	<b>Prolog</b>	<b>4</b>
<b>3</b>	<b>Parallelprogrammierung</b>	<b>5</b>
3.1	Grundlagen . . . . .	5
3.1.1	Coffman-Bedingungen (Deadlock-Bedingungen) (Folie: 54_(47)) . . . . .	5
3.1.2	Flynn's Taxonomy (Folie: 51_(13)) (Grafiken von SS 14, Nr. 6) . . . . .	5
3.1.3	Beschleunigung: Amdahl's Law (Folie: 51_(17)) . . . . .	6
3.2	MPI . . . . .	6

# Kapitel 1

## Theoretische Grundlagen

### 1.1 Äquivalenz

#### 1.1.1 $\alpha$ -Äquivalenz (Folie: 20\_(166))

gleicher Ausdruck / Funktion, nur andere Namen  $\Rightarrow$  durch Umbenennung Transformation von  $t_1$  zu  $t_2$  möglich

#### 1.1.2 $\eta$ -Äquivalenz (Folie: 20\_(167))

Zwei Funktionen sind gleich, falls Ergebnis gleich für alle Argumente

#### 1.1.3 Church-Zahlen (Folie: 20\_(179ff))

- Zahlen:

$$\begin{aligned}c_0 &= \lambda s. \lambda z. z \\c_1 &= \lambda s. \lambda z. s z \\c_2 &= \lambda s. \lambda z. s (s z) \\&\vdots \\c_n &= \lambda s. \lambda z. s^n z\end{aligned}$$

- Operationen

- Nachfolgerfunktion  $succ = \lambda n. \lambda s. \lambda z. s (n s z)$
- Addition  $plus = \lambda m. \lambda n. \lambda s. \lambda z. m s (n s z)$
- Multiplikation  $times = \lambda m. \lambda n. \lambda s. n (m s) \stackrel{\eta}{=} \lambda m. \lambda n. \lambda s. \lambda z. n (m s) z$
- Potenzieren  $exp = \lambda m. \lambda n. n m \stackrel{\eta}{=} \lambda m. \lambda n. \lambda s. \lambda z. n m s z$
- Subtraktion  $sub = \lambda m. \lambda n. n pred m$  (ÜB 5, Nr. 4)
  - \*  $pair = \lambda a. \lambda b. \lambda f. f a b$
  - \*  $fst = \lambda p. p(\lambda a. \lambda b. a)$
  - \*  $snd = \lambda p. p(\lambda a. \lambda b. b)$
  - \*  $next = \lambda p. pair (snd p) (succ (snd p))$
  - \*  $pred = \lambda n. fst (n next (pair c_0 c_0))$
- Vergleichsoperation „lessEq  $m \leq n$ “  $lessEq = \lambda m. \lambda n. isZero (sub m n)$  (ÜB 6, Nr. 2)
- Vergleichsoperation „greaterEq  $m \geq n$ “  $greaterEq = \lambda m. \lambda n. isZero (sub n m)$
- Vergleichsoperation „eq  $m == n$ “  $eq = \lambda m. \lambda n. (\lambda a. a) (lessEq m n) (greaterEq m n) c_{false}$
- $isZero = \lambda n. n (\lambda p. c_{false}) c_{true}$  (SS 14)

- boolesche Werte

- True  $c_{true} = \lambda t. \lambda f. t$
- False  $c_{false} = \lambda t. \lambda f. f$

### 1.1.4 Rekursionsoperator Y (Folie: 20\_(184))

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

- Rekursionsoperator Y ist nicht typisierbar (Folie: 21\_(205))

## 1.2 Typinferenz

Bei Let-Polymorphismus angepasste Regeln von VAR und ABS (Folie: 22\_(211))

Regel	Constraint	Bsp.
$\text{CONST} \frac{c \in \text{Const}}{\Gamma \vdash c : \tau_c}$	$\text{CONST} \frac{c \in \text{Const}}{\Gamma \vdash c : \alpha_1} : \alpha_1 = \text{Typ}(c)$	$\text{CONST} \frac{1 \in \text{Const}}{\Gamma \vdash 1 : \alpha_1} : \alpha_1 = \text{Typ}(1) = \text{Int}$
$\text{VAR} \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$	$\text{VAR} \frac{(x : \alpha_1)(x) = \alpha_2}{(x : \alpha_1) \vdash x : \alpha_2} : \alpha_1 = \alpha_2$	
$\text{ABS} \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x.t : \tau_1 \rightarrow \tau_2}$	$\text{ABS} \frac{\Gamma, x : \alpha_2 \vdash t : \alpha_3}{\Gamma \vdash \lambda x.t : \alpha_1} : \alpha_1 = \alpha_2 \rightarrow \alpha_3$	
$\text{APP} \frac{\Gamma \vdash t_1 : \tau_2 \rightarrow \tau \quad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash t_1\ t_2 : \tau}$	$\text{APP} \frac{\Gamma \vdash t_1 : \alpha_2 \quad \Gamma \vdash t_2 : \alpha_3}{\Gamma \vdash t_1\ t_2 : \alpha_1} : \alpha_2 = \alpha_3 \rightarrow \alpha_1$	
$\text{LET} \frac{\Gamma \vdash t_1 : \tau_1 \quad \Gamma, x : ta(\tau_1, \Gamma) \vdash t_2 : \tau_2}{\Gamma \vdash \text{let } x=t_1 \text{ in } t_2 : \tau_2}$	$\text{LET} \frac{\Gamma \vdash t_1 : \alpha_2 \quad \Gamma, x : ta(\alpha_2, \Gamma) \vdash t_2 : \alpha_3}{\Gamma \vdash \text{let } x=t_1 \text{ in } t_2 : \alpha_1} : \alpha_2 = \alpha_3$	

# Kapitel 2

## Prolog

- Vergleich arithmetischer Ausdrücke
  - gleich: „ $=$ “
  - ungleich: „ $\neq$ “
  - kleiner: „ $<$ “
  - kleiner-gleich : „ $\leq$ “
  - größer: „ $>$ “
  - größer-gleich: „ $\geq$ “

# Kapitel 3

## Parallelprogrammierung

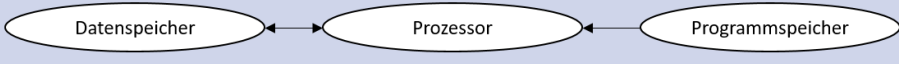
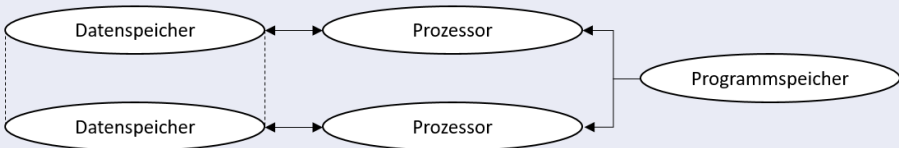
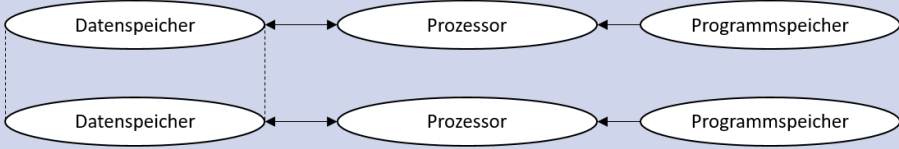
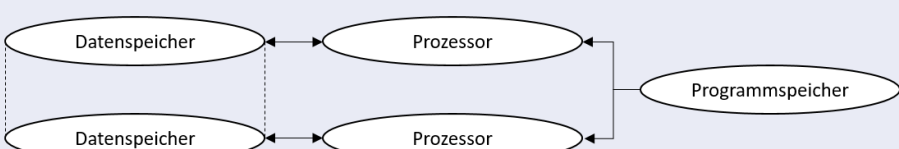
### 3.1 Grundlagen

#### 3.1.1 Coffman-Bedingungen (Deadlock-Bedingungen) (Folie: 54\_(47))

Wenn alle vier der folgenden Bedingungen zutreffen, liegt ein Deadlock vor. Deadlocks können verhindert werden, indem immer mindestens eine Bedingung nicht erfüllt ist, d.h. nicht alle auf einmal erfüllt sein können.

1. **Mutual exclusion**
  - beschränkter Zugriff auf eine Ressource
  - Ressource kann nur mit einer beschränkten Anzahl von Nutzern geteilt werden
2. **Hold and wait**: Warten auf alle benötigten Ressourcen, während die Kontrolle über bisher zugesprochene (mind. eine) Ressourcen behalten wird.
3. **No preemption**: Zugewiesene Ressourcen können nur freiwillig zurückgegeben werden, die Rückgabe kann nicht erzwungen werden.
4. **Circular Wait**: Möglichkeit von Kreisen in Ressourcen-Anfragen Graph:  
Zyklische Kette von Prozessen, die bereits Ressourcen (mind. eine) erhalten haben und gleichzeitig auf weitere Ressourcen warten, welche jeweils dem nächsten Prozess in der zirkulären Kette zugesprochen wurden.

#### 3.1.2 Flynn's Taxonomy (Folie: 51\_(13)) (Grafiken von SS 14, Nr. 6)

Taxonomie		Graphische Repräsentation
SISD (Single Instruction x Single Data)	von Neumann Architektur: eine Anweisungsreihenfolge auf einem Speicher ausgeführt	
SIMD (Single Instruction x Multiple Data)	Eine Anweisung auf homogene Daten angewendet (z.B. Array) z.B. Vektor Prozessoren von frühen Supercomputern	
MIMD (Multiple Instruction x Multiple Data)	versch. Prozessoren operieren auf versch. Daten z.B. aktuelle Mehrkernprozessoren	
MISD (Multiple Instruction x Single Data)	Mehrere Anweisungen gleichzeitig auf gleichen Daten ausgeführt z.B. redundante Architekturen oder Pipelines in modernen Prozessoren	

### 3.1.3 Beschleunigung: Amdahl's Law (Folie: 51\_(17))

Beschleunigung eines Algo durch die Verwendung von n Prozessoren:

$$S(n) = \frac{T(1)}{T(n)} = \frac{\text{Ausführungszeit mit einem Prozessor}}{\text{Ausführungszeit mit n Prozessoren}}$$

maximale Beschleunigung durch parallele Ausführung mit n Prozessoren:

$$S(n) = \frac{1}{(1-p) + \frac{p}{n}}$$

p: parallelisierbarer prozentualer Anteil des Programms

## 3.2 MPI

- `MPI_Comm_WORLD`: Standard Communicator
- `MPI_Init(&argc, &args)`: initialisiere MPI
- `MPI_Finalize()`: Clean-Up nach Ausführung von MPI
- `MPI_Comm_rank(MPI_Comm_WORLD, &my_rank)`: `my_rank` enthält den Rank des aktuellen Prozesses
- `MPI_Comm_size(MPI_Comm_WORLD, &size)`: `size` enthält Gesamtanzahl von Prozessen
- `MPI_Barrier(MPI_Comm_WORLD)`: Barriere, blockiert bis alle Prozesse die Barriere aufgerufen haben
- `MPI_Send(void* buffer, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
- `MPI_Recv(void* buffer, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status* status)`

## 3.3 Java