# Tutorium 07: Prolog, Typinferenz

Paul Brinkmeier

15. Dezember 2020

Tutorium Programmierparadigmen am KIT

# Heutiges Programm

# Programm

- Typinferenz
- Aufgaben zu Prolog

# Typinferenz

# Unifikation (Robinson, [Rob65])

## Unifikationsalgorithmus: $\texttt{unify}(C) =$

```
if C == ∅ then []
else let {θ_l = θ_r} ∪ C' = C in
  if θ_l == θ_r then unify(C')
  else if θ_l == Y and Y ∉ FV(θ_r) then unify([Y ↦ θ_r] C') ∘ [Y ↦ θ_r]
  else if θ_r == Y and Y ∉ FV(θ_l) then unify([Y ↦ θ_l] C') ∘ [Y ↦ θ_l]
  else if θ_l == f(θ_l^1,...,θ_l^n) and θ_r == f(θ_r^1,...,θ_r^n)
       then unify(C' ∪ {θ_l^1 = θ_r^1,...,θ_l^n = θ_r^n})
  else fail
```

$\texttt{Y} \in FV(\theta)$ occur check, verhindert zyklische Substitutionen

## Korrektheitstheorem

$\texttt{unify}(C)$ terminiert und gibt *mgu* für $C$ zurück, falls $C$ unifizierbar, ansonsten **`fail`**.

Beweis: Siehe [Pie02]

# Prolog-Unifikation

Unifiziert:

- `A = x`
- `B = f(x)`
- `C = g(C)`
- `f(x, A, z) = f(x, y, B)`
- `func(A, B, z) = func(x, y, A)`
- `g(x, A, z) = f(x, A, A)`
- `f(g(z)) = f(D)`

Ergebnis: Entweder **fail** oder ein Unifikator.

# Cheatsheet: Lambda-Kalkül/Basics

- Terme $t$: Variable ($x$), Funktion ($\lambda x.t$), Anwendung ($t\ t$)
- $\alpha$-Äquivalenz: Gleiche Struktur
- $\eta$-Äquivalenz: Unterversorgung
- Freie Variablen, Substitution, RedEx
- $\beta$-Reduktion:
  $(\lambda p.b)\ t \Rightarrow b\,[p \to t]$

# Cheatsheet: Typisierter Lambda-Kalkül

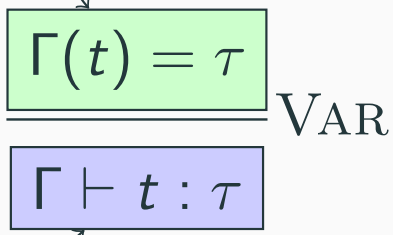
$$\frac{\Gamma(t) = \tau}{\Gamma \vdash t : \tau}\textsc{Var} \qquad \frac{\Gamma \vdash f : \phi \to \alpha \qquad \Gamma \vdash x : \phi}{\Gamma \vdash f\ x : \alpha}\textsc{App}$$

$$\frac{\Gamma, p : \pi \vdash b : \rho}{\Gamma \vdash \lambda p.\, b : \pi \to \rho}\textsc{Abs}$$

- Typvariablen: $\tau$, $\alpha$, $\pi$, $\rho$
- Funktionstypen: $\tau_1 \to \tau_2$, rechtsassoziativ
- (Weitere Typen: Listen, Tupel, etc.)
- Typisierungsregeln sind eindeutig: Eine Regel pro Termform

# (Allgemeine) Typisierungsregel für Variablen

- „Der Typkontext $\Gamma$ enthält einen Typ $\tau$ für $t$.“




- Daraus folgt:
- „Variable $t$ hat im Kontext $\Gamma$ den Typ $\tau$.“

# Typisierungsregel für Funktionsanwendungen

- „$f$ ist im Kontext $\Gamma$ eine Funktion, die $\phi$s auf $\alpha$s abbildet.“
- „$x$ ist im Kontext $\Gamma$ ein Term des Typs $\phi$.“

$$\frac{\Gamma \vdash f : \phi \to \alpha \qquad \Gamma \vdash x : \phi}{\Gamma \vdash f\ x : \alpha}\ \textsc{App}$$

- Daraus folgt:
- „$x$ eingesetzt in $f$ ergibt einen Term des Typs $\alpha$.“

# Typisierungsregel für Lambdas

- „Unter Einfügung des Typs $\pi$ von $p$ in den Kontext...“
- „... ist $b$ als Funktion von $p$ typisierbar.“

$$\frac{\Gamma, p : \pi \qquad \vdash b : \rho}{\Gamma \vdash \lambda p.\, b : \pi \to \rho}\ \textsc{Abs}$$

- Daraus folgt:
- „$\lambda p.\, b$ ist eine Funktion, die $\pi$s auf $\rho$s abbildet“

# Typinferenz

Vorgehensweise zur Typinferenz:

- Stelle Typherleitungsbaum auf
  - In jedem Schritt werden neue Typvariablen $\alpha_i$ angelegt
  - Statt die Typen direkt im Baum einzutragen, werden Gleichungen in einem Constraint-System eingetragen
- Unifiziere Constraint-System zu einem Unifikator
  - Robinson-Algorithmus, im Grunde wie bei Prolog
  - I.d.R.: Allgemeinster Unifikator (findet man per Robinson)

# Unifikation (Robinson, [Rob65])

## Unifikationsalgorithmus: $\texttt{unify}(C) =$

```
if C == ∅ then []
else let {θ_l = θ_r} ∪ C' = C in
  if θ_l == θ_r then unify(C')
  else if θ_l == Y and Y ∉ FV(θ_r) then unify([Y ↦ θ_r] C') ∘ [Y ↦ θ_r]
  else if θ_r == Y and Y ∉ FV(θ_l) then unify([Y ↦ θ_l] C') ∘ [Y ↦ θ_l]
  else if θ_l == f(θ_l^1,...,θ_l^n) and θ_r == f(θ_r^1,...,θ_r^n)
       then unify(C' ∪ {θ_l^1 = θ_r^1,...,θ_l^n = θ_r^n})
  else fail
```

$\texttt{Y} \in FV(\theta)$ occur check, verhindert zyklische Substitutionen

## Korrektheitstheorem

$\texttt{unify}(C)$ terminiert und gibt *mgu* für $C$ zurück, falls $C$ unifizierbar, ansonsten **`fail`**.

Beweis: Siehe [Pie02]

# Unifikation

## Unifikationsalgorithmus: $\texttt{unify}(C) =$

```
if C == ∅ then []
else let {τ1 = τ2} ∪ C' = C in
  if τ1 == τ2 then unify(C')
  else if τ1 == α and α ∉ FV(τ2) then unify([α ↦ τ2] C') ∘ [α ↦ τ2]
  else if τ2 == α and α ∉ FV(τ1) then unify([α ↦ τ1] C') ∘ [α ↦ τ1]
  else if τ1 == (τ1' → τ1'') and τ2 == (τ2' → τ2'')
       then unify(C' ∪ {τ1' = τ2', τ1'' = τ2''})
  else fail
```

$\alpha \in FV(\tau)$ occur check, verhindert zyklische Substitutionen

## Korrektheitstheorem

$\texttt{unify}(C)$ terminiert und gibt *mgu* für $C$ zurück, falls $C$ unifizierbar, ansonsten `fail`.

Beweis: Siehe Literatur

# Unifikation für Typinferenz

Typen kann man auch als Funktoren darstellen:

$$\tau_1 \to \tau_2 \equiv \texttt{func}(\tau_1, \tau_2)$$

$$[\tau] \equiv \texttt{list}(\tau)$$

etc.

# Typinferenz: Übungsaufgaben

$$\frac{\cdots}{f : \text{int} \to \beta \vdash \lambda x.\, f\ x : \alpha_1}\,\textsc{Abs}$$

- „Finde den allgemeinsten Typen $\alpha_1$ von $\lambda x.\, f\ x$“

Erinnerung:

- Baum mit durchnummerierten $\alpha_i$ aufstellen
- Constraints sammeln:

$$\frac{\Gamma(t) = \alpha_j}{\Gamma \vdash t : \alpha_i}\,\textsc{Var}$$

Constraint:
$\{\alpha_i = \alpha_j\}$

$$\frac{\Gamma \vdash f : \alpha_j \qquad \Gamma \vdash x : \alpha_k}{\Gamma \vdash f\ x : \alpha_i}\,\textsc{App}$$

Constraint:
$\{\alpha_j = \alpha_k \to \alpha_i\}$

$$\frac{\Gamma, p : \alpha_j \vdash b : \alpha_k}{\Gamma \vdash \lambda p.\, b : \alpha_i}\,\textsc{Abs}$$

Constraint:
$\{\alpha_i = \alpha_j \to \alpha_k\}$

- Constraint-System auflösen

# Typinferenz: Übungsaufgaben

$$\frac{\dots}{\vdash \lambda f.\, \lambda x.\, (f\ x)\ x : \alpha_1}\,\textsc{Abs}$$

- „Finde den allgemeinsten Typen $\alpha_1$ von $\lambda f.\, \lambda x.\, (f\ x)\ x$“

Erinnerung:

- Baum mit durchnummerierten $\alpha_i$ aufstellen
- Constraints sammeln:

$$\frac{\Gamma(t) = \alpha_j}{\Gamma \vdash t : \alpha_i}\,\textsc{Var} \qquad \text{Constraint: } \{\alpha_i = \alpha_j\}$$

$$\frac{\Gamma \vdash f : \alpha_j \qquad \Gamma \vdash x : \alpha_k}{\Gamma \vdash f\ x : \alpha_i}\,\textsc{App} \qquad \text{Constraint: } \{\alpha_j = \alpha_k \to \alpha_i\}$$

$$\frac{\Gamma, p : \alpha_j \vdash b : \alpha_k}{\Gamma \vdash \lambda p.\, b : \alpha_i}\,\textsc{Abs} \qquad \text{Constraint: } \{\alpha_i = \alpha_j \to \alpha_k\}$$

- Constraint-System auflösen

# Prolog

# Cheatsheet: Prolog

- Terme:
  - Variablen: `Var`, `X`, `X2`
  - Funktoren/Atome: `f(a, b, c)`, `app(f, x)`, `main`
  - Arithmetische Ausdrücke: `17 + 25`, `6 * 7`
- Regeln: `rule(P1, ..., PN) :- Goal1, ..., GoalM.`
- Ziele:
  - Funktor: `member(X, [1,2,3])`
  - Unifikation: `X = Y`
  - Arithmetik: `N is M + 1`
  - Verneinung: `not(G)`
  - Arithmetischer Vergleich: `X =:= Y`, `X =\= Y`, etc.
  - Cut: `!`
- Konzepte: Unifikation, Resolution

# Prolog — Regelsysteme als Programmiersprache

```
grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
parent(X, Y) :- mother(X, Y).
parent(X, Y) :- father(X, Y).

mother(inge, emil).
mother(inge, petra).
father(emil, kunibert).
```

`?- grandparent(inge, kunibert).` $\rightsquigarrow$ `yes.`

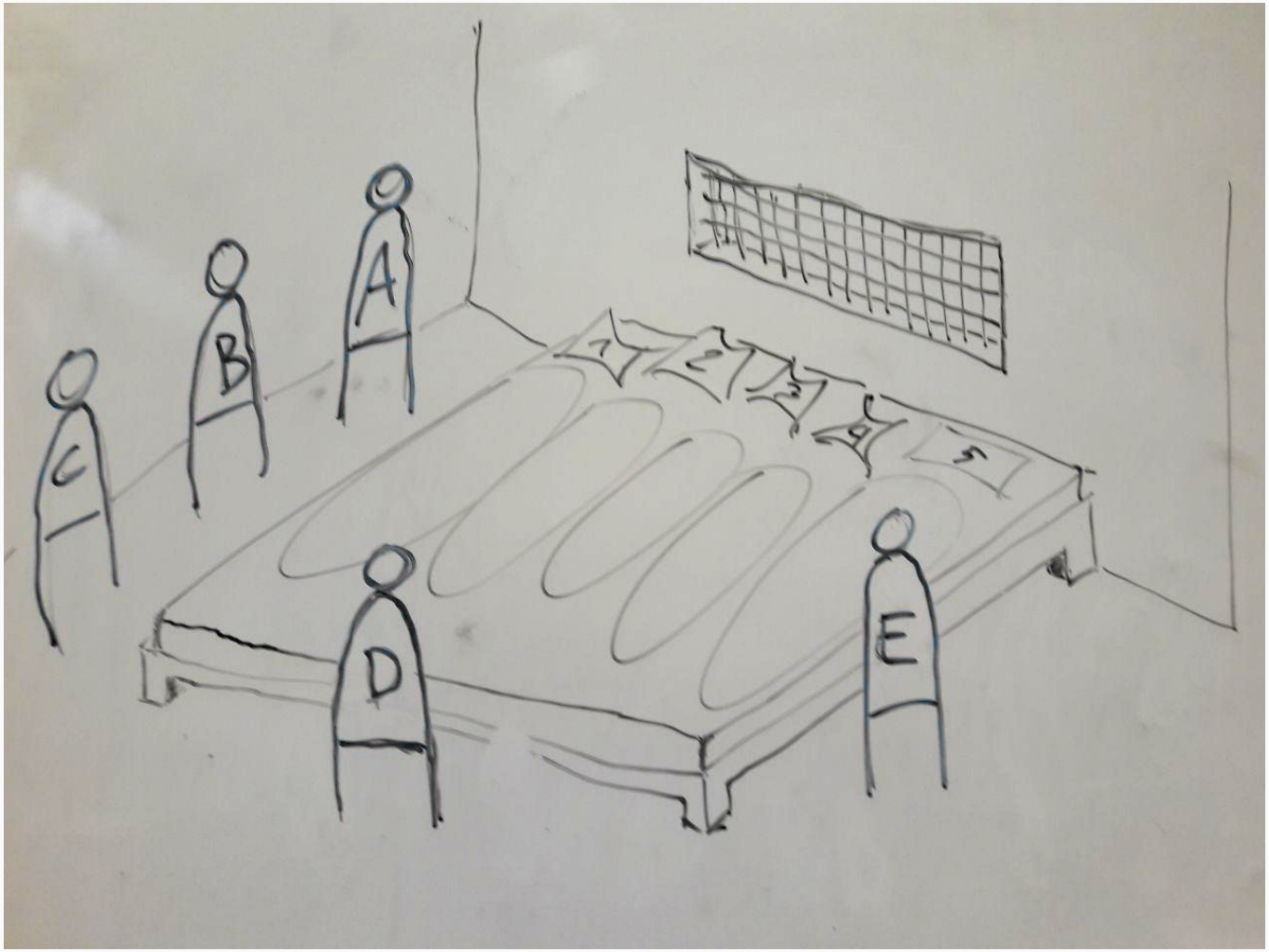
# Prolog — Regelsysteme als Programmiersprache

```
grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
parent(X, Y) :- mother(X, Y).
parent(X, Y) :- father(X, Y).

mother(inge, emil).
mother(inge, petra).
father(emil, kunibert).
```

$$\dfrac{\dfrac{\dfrac{}{\texttt{mother(inge, emil)}}}{\texttt{parent(inge, emil)}} \quad \dfrac{\dfrac{}{\texttt{father(emil, kunibert)}}}{\texttt{parent(emil, kunibert)}}}{\texttt{grandparent(inge, kunibert)}}$$

# Schlafplätze im Gefängnis



# Dinesman's multiple-dwelling problem

Bob kommt nun ins Gefängnis. Aaron, Bob, Connor, David und Edison müssen sich zu fünft ein sehr breites Bett teilen.

- Aaron will nicht am rechten Ende liegen
- Bob will nicht am linken Ende liegen
- Connor will an keinem der beiden Enden liegen
- David will weiter rechts liegen als Bob
- Connor schnarcht sehr laut;
  Bob und Edison sind sehr geräuschempfindlich
  - $\rightsquigarrow$ Bob will nicht direkt neben Connor liegen
  - $\rightsquigarrow$ Edison will nicht direkt neben Connor liegen

Wie können die 5 Schlafplätze verteilt werden?

# Schlafplätze im Gefängnis

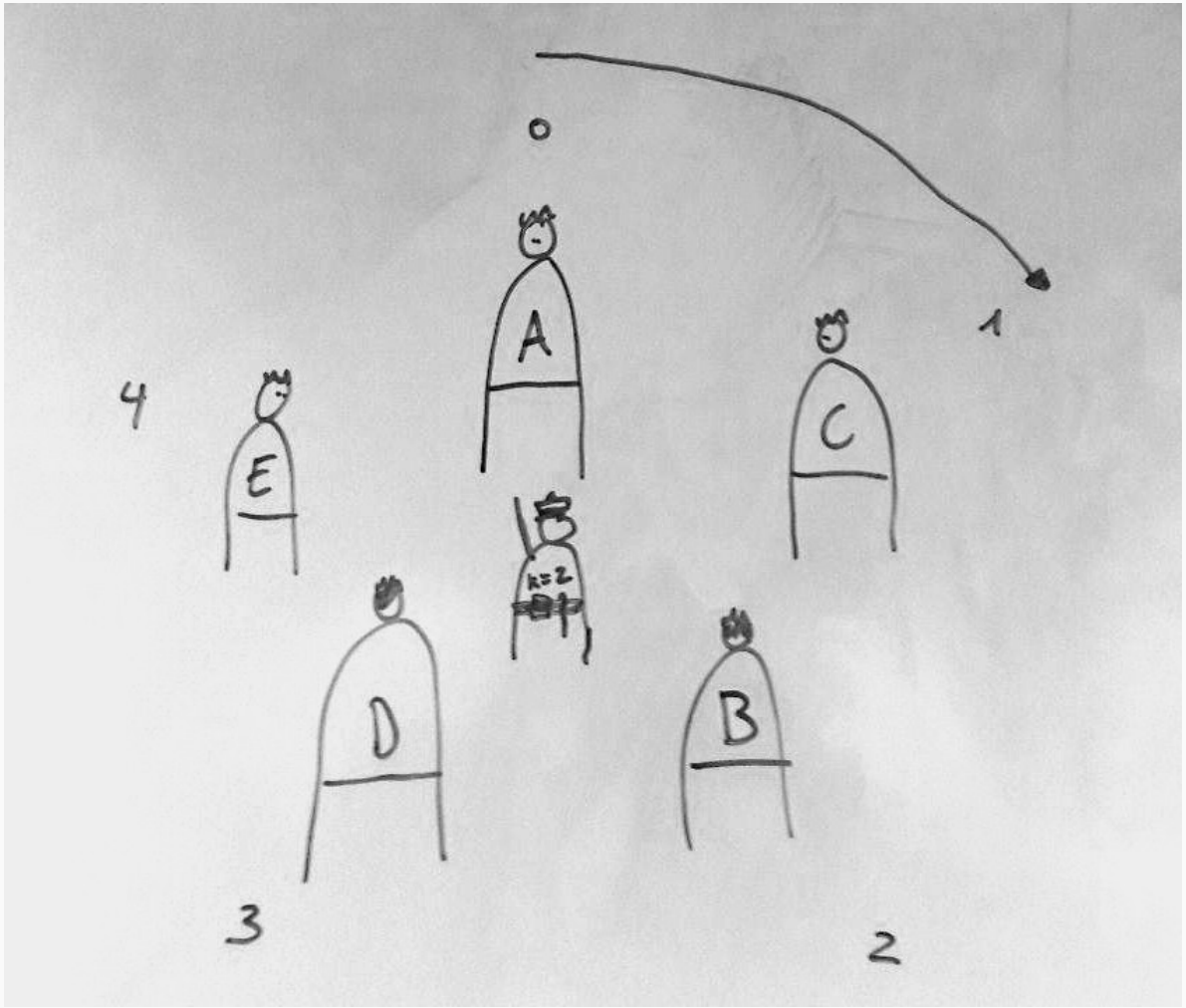
```
% schlafplaetze.pl

bett(X) :- member(X, [1, 2, 3, 4, 5]).

schlafplaetze(A, B, C, D, E) :-
  bett(A), bett(B), bett(C), bett(D), bett(E),
  distinct([A, B, C, D, E]),
  % weitere Tests
```

- Fügt weitere benötigte Tests ein
- Implementiert:
  - `distinct/1` prüft Listenelemente auf paarweise Ungleichheit
  - `adjacent/2` prüft, ob $|A - B| = 1$

# Putzdienst



# Putzdienst

- Aaron, Bob, Connor, David und Edison sollen 4 Einheiten Putzdienst übernehmen
- Da sie sich nicht einigen können, wer aussetzen darf, wendet ein Wärter folgendes Vorgehen an:
  - Die fünf werden im Kreis aufgestellt
  - Der Wärter stellt sich in die Mitte
  - Beginnend bei 12 Uhr dreht er sich im Uhrzeigersinn und teilt jeden $k$-ten (bspw. $k = 2$) Insassen zum Putzdienst ein
    - D.h. es werden immer $k - 1$ Insassen übersprungen

An welcher Stelle muss Bob stehen, um nicht putzen zu müssen?

# Putzdienst

- Aaron, Bob, Connor, David und Edison sollen 4 Einheiten Putzdienst übernehmen
- Da sie sich nicht einigen können, wer aussetzen darf, wendet ein Wärter folgendes Vorgehen an:
  - Die fünf werden im Kreis aufgestellt
  - Der Wärter stellt sich in die Mitte
  - Beginnend bei 12 Uhr dreht er sich im Uhrzeigersinn und teilt jeden $k$-ten (bspw. $k = 2$) Insassen zum Putzdienst ein
    - D.h. es werden immer $k - 1$ Insassen übersprungen

An welcher Stelle muss Bob stehen, um nicht putzen zu müssen?

An welcher Stelle muss Bob bei 41 Insassen und $k = 3$ stehen?

# Putzdienst

```
% putzdienst.pl

% Bspw.
% ?- keinPutzdienstFuer([a, b, c, d, e], 2, X)
keinPutzdienstFuer(L, K, X) :-
  Countdown is K - 1,
  helper(L, Countdown, K, X).

helper([X], _C, _K, X) :- !.
...
```

- Weitere Fälle für `helper/4`:
  - `C = 0` $\rightsquigarrow$ Element entfernen
  - Ansonsten: Element hinten wieder anhängen

# Quellen der Aufgaben

Zum Nachlesen und Vergleichen mit Lösungen in anderen Programmiersprachen:

- WG — Rosetta Code: Department Numbers
- Detektiv — github.com/Anniepoo/prolog-examples
- Schlafplätze — SICP, S. 418
- Putzdienst — Rosetta Code: Josephus problem

# Schöne Ferien!

# Schöne Ferien!

- Nach dem Ferien: Mehr Typinferenz, Reussner-Teil
- Aktueller Klausurtermin:
  09.04.2021, 17:00, Zelt auf dem Forum

Bleibt gesund, feiert schön und einen guten Rutsch!