Tutorium 07: Unifikation

Paul Brinkmeier

13. Dezember 2022

Tutorium Programmierparadigmen am KIT

Unifikation

Welche Probleme löst die Unifikation?

$$\texttt{state}(\texttt{l},\texttt{l},\texttt{l},\texttt{l}) \stackrel{!}{=} \texttt{state}(M,W,Z,K)$$
$$\texttt{opposite}(M,M_2) \stackrel{!}{=} \texttt{opposite}(\texttt{l},\texttt{r})$$
$$Z \stackrel{!}{=} K$$

- Unifikation löst Gleichungssysteme von baumförmigen Termen (hier: Prolog-Terme).
- Eingabe: Menge $C = \{\theta_l^1 \stackrel{!}{=} \theta_r^1, ..., \theta_l^n \stackrel{!}{=} \theta_r^n\}$
  - Alle $\theta^i_{\{l,r\}}$ sind Bäume und können Variablen enthalten.
- Ausgabe: *Unifikator* σ, sodass $\sigma(\theta_l^i) = \sigma(\theta_r^i)$.
  - Wenn C nicht unifizierbar (bspw. $X = \texttt{f}(X)$): `fail`

Welche Probleme löst die Unifikation?

$$\texttt{state}(\texttt{l}, \texttt{l}, \texttt{l}, \texttt{l}) \stackrel{!}{=} \texttt{state}(M, W, Z, K)$$
$$\texttt{opposite}(M, M_2) \stackrel{!}{=} \texttt{opposite}(\texttt{l}, \texttt{r})$$
$$Z \stackrel{!}{=} K$$

Mehrere mögliche Lösungen:

$$\sigma_1 = [M_2 \mapsto \texttt{r}] \circ [K \mapsto \texttt{l}] \circ [Z \mapsto \texttt{l}] \circ [W \mapsto \texttt{l}] \circ [M \mapsto \texttt{l}]$$
$$\sigma_2 = [M_2 \mapsto \texttt{r}] \circ [K \mapsto \texttt{l}] \circ [Z \mapsto \texttt{K}] \circ [W \mapsto \texttt{l}] \circ [M \mapsto \texttt{l}]$$

- I.d.R. suchen wir nach *einem* allgemeinsten Unifikator (mgu).
- mgu $\approx$ minimaler Unifikator, der C löst.

Prolog-Unifikation

Unifiziert:

$$A \stackrel{!}{=} \mathtt{x}$$

$$B \stackrel{!}{=} \mathtt{f}(X)$$

$$C \stackrel{!}{=} \mathtt{g}(C)$$

$$\mathtt{f}(\mathtt{x}, D, \mathtt{z}) \stackrel{!}{=} \mathtt{f}(\mathtt{x}, \mathtt{y}, E)$$

$$\mathtt{func}(F, \mathtt{func}(G, \mathtt{z})) \stackrel{!}{=} \mathtt{func}(\mathtt{x}, \mathtt{func}(\mathtt{y}, F))$$

$$\mathtt{g}(\mathtt{x}, H, \mathtt{z}) \stackrel{!}{=} \mathtt{f}(\mathtt{x}, H, H)$$

$$\mathtt{f}(\mathtt{g}(\mathtt{z})) \stackrel{!}{=} \mathtt{f}(J)$$

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

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]

Robinson-Algorithmus: Zeile für Zeile

$$\texttt{if } C == \emptyset \texttt{ then } []$$
$$\texttt{else let } \{\theta_l \stackrel{!}{=} \theta_r\} \cup C' = C \texttt{ in}$$

- Ist das Gleichungssystem C leer, ist es schon gelöst
  $\leadsto$ wir brauchen nichts zu ersetzen.
- Andernfalls betrachten wir *eine* der Gleichungen: $\theta_l \stackrel{!}{=} \theta_r$.
  - *Beliebige Auswahl möglich*.
  - Die restlichen Gleichungen merken wir uns als C'.
- Beispiel:
  $C = \{X \stackrel{!}{=} \mathtt{a}, Y \stackrel{!}{=} \mathtt{f}(X), \mathtt{f}(Z) \stackrel{!}{=} Y\}$
  $\theta_l = X, \theta_r = \mathtt{a}, C' = \{Y \stackrel{!}{=} \mathtt{f}(X), \mathtt{f}(Z) \stackrel{!}{=} Y\}$

Robinson-Algorithmus: Zeile für Zeile

$$\texttt{if } \theta_l == \theta_r \texttt{ then unify}(C')$$

- Wenn die Gleichung trivial ist (auf beiden Seiten steht schon das gleiche), brauchen wir auch nichts zu ersetzen.
- Wir müssen also nur C' unifizieren.
- Verschiedene Gleichheitsrelationen:
  - $A \stackrel{!}{=} B$: Element von C, behandeln wir wie eine Datenstruktur.
  - $A == B$: Vergleichsoperator
  - $A = B$: Meta-Gleichheitsoperator, Notation für Pattern-Matching

Robinson-Algorithmus: Zeile für Zeile

$$\texttt{else if } \theta_l == Y \texttt{ and } Y \notin FV(\theta_r)$$
$$\texttt{then unify}([Y \Rightarrow \theta_r]\, C') \circ [Y \Rightarrow \theta_r]$$

- Steht auf der linken Seite eine Variable, so wird diese ersetzt.
  - $\theta_l == Y$: „Ist der Term θ_l eine Variable Y?“
  - $Y \notin FV(\theta_r)$: *occurs check*, Y darf sich nicht selbst einsetzen.
- Wir ersetzen in C' dann Y durch θ_r.
- Substitution $[Y \Rightarrow \theta_r]$ wird als Ergebnis vorgemerkt.
- Beispiel: $\theta_l = X$ ✓, $X \notin FV(\theta_r) = FV(\mathtt{a}) = \emptyset$ ✓, d.h.
  Ergebnis: $\mathtt{unify}(\{Y \stackrel{!}{=} \mathtt{f}(\mathtt{a}), \mathtt{f}(Z) \stackrel{!}{=} Y\}) \circ [X \Rightarrow \mathtt{a}]$

Robinson-Algorithmus: Zeile für Zeile

$$\texttt{else if } \theta_r == Y \texttt{ and } Y \notin FV(\theta_l)$$
$$\texttt{then unify}([Y \Rightarrow \theta_l]\, C') \circ [Y \Rightarrow \theta_l]$$

- Auch wenn rechts eine Variable steht muss sie ersetzt werden.
- Beispiel: $\theta_r = Y$ ✓, $Y \notin FV(\theta_l) = FV(\texttt{f}(Z)) = \{Z\}$ ✓, d.h. Ergebnis: $\texttt{unify}(\{\texttt{f}(Z) \stackrel{!}{=} \texttt{f}(\texttt{a})\}) \circ [Y \Rightarrow f(Z)]$

Robinson-Algorithmus: Zeile für Zeile

$$\texttt{else if } \theta_l == \mathtt{f}(\theta_l^1, ..., \theta_r^n) \texttt{ and } \theta_r == \mathtt{f}(\theta_r^1, ..., \theta_r^n)$$
$$\texttt{then unify}(C' \cup \{\theta_l^1 \stackrel{!}{=} \theta_r^1, ..., \theta_l^n \stackrel{!}{=} \theta_r^n\})$$

- Steht auf beiden Seiten ein Funktor, extrahieren wir paarweise neue Gleichungen und unifizieren diese mitsamt C'.
  - Namen der Funktoren *müssen identisch sein*! (hier: `f`)
  - Parameterzahlen der Funktoren *müssen identisch sein*!
  - Für Atome: $n = 0$, aber schon abgedeckt durch den ersten Fall.
- Beispiel: $\theta_l = \mathtt{f}(Z), \theta_r = \mathtt{f}(\mathtt{a})$ ✓, $C' = \emptyset$
  Ergebnis: $\texttt{unify}(\emptyset \cup \{Z \stackrel{!}{=} \mathtt{a}\}) = [Z \Rightarrow \mathtt{a}]$ (links Variable)

Typinferenz

Struktur von Lambda-Termen

Lambda-Terme bestehen aus

- Lambdas $\lambda \mathtt{p}.\, b$
- Funktionsanwendungen $x\ y$
- Variablen x, Konstanten $\mathtt{true}, 17$

$$\lambda \mathtt{a}.\, \lambda \mathtt{f}.\, f\ (a\ \mathtt{true})$$

Struktur von Lambda-Termen

Lambda-Terme bestehen aus

- **Lambdas** $\lambda \mathrm{p}.\, b$
- Funktionsanwendungen $x\ y$
- Variablen x, Konstanten $\texttt{true}, 17$

$$\underbrace{\lambda \mathrm{a}.\ \underbrace{\lambda \mathtt{f}.\, f\ (a\ \mathtt{true})}_{\substack{\text{Lambda} \\ p{=}f, b{=}f\ (a\ \mathtt{true})}}}_{\substack{\text{Lambda} \\ p{=}a, b{=}\lambda \mathtt{f}.\, f\ (a\ \mathtt{true})}}$$

Struktur von Lambda-Termen

Lambda-Terme bestehen aus

- Lambdas $\lambda \mathrm{p}.\, b$
- **Funktionsanwendungen** $x\ y$
- Variablen x, Konstanten $\texttt{true}, 17$

$$\lambda \mathrm{a}.\, \lambda \mathrm{f}.\, \underbrace{f\ \underbrace{(a\ \texttt{true})}_{\substack{\text{Anwendung} \\ x=a, y=\texttt{true}}}}_{\substack{\text{Anwendung} \\ x=f, y=a\ \texttt{true}}}$$

Struktur von Lambda-Termen

Lambda-Terme bestehen aus

- Lambdas $\lambda \mathtt{p}.\, b$
- Funktionsanwendungen $x\ y$
- **Variablen** x, Konstanten $\mathtt{true}, 17$

$$\lambda \mathtt{a}.\, \lambda \mathtt{f}.\ \underbrace{f}_{\text{Variable}}\ (\ \underbrace{a}_{\text{Variable}}\ \ \mathtt{true})$$

Struktur von Lambda-Termen

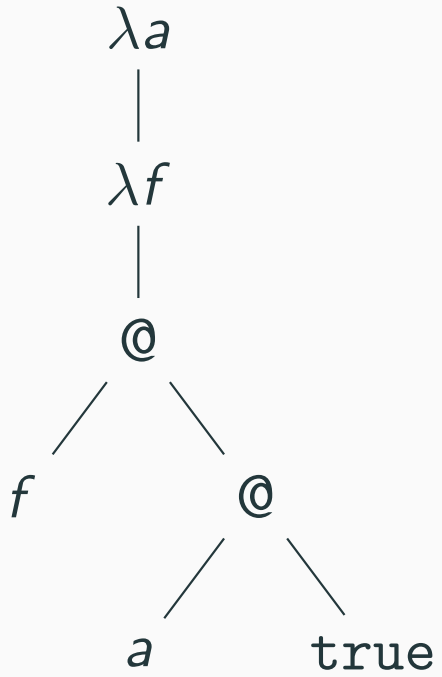
Lambda-Terme bestehen aus

- Lambdas $\lambda \mathrm{p}.\, b$
- Funktionsanwendungen $x\ y$
- Variablen x, **Konstanten** $\texttt{true}, 17$

$$\lambda \mathrm{a}.\, \lambda \mathrm{f}.\, f\ (a\ \underbrace{\texttt{true}}_{\text{Konstante}}\ )$$

Lambda-Terme als Bäume

Wir können Lambda-Terme also als Bäume mit Lambda- und Anwendungsknoten und Variablen- und Konstantenblättern betrachten, um ihre Struktur zu untersuchen:



$\lambda \mathtt{a}.\, \lambda \mathtt{f}.\, f\ (a\ \mathtt{true})$

Cheatsheet: Typisierter Lambda-Kalkül

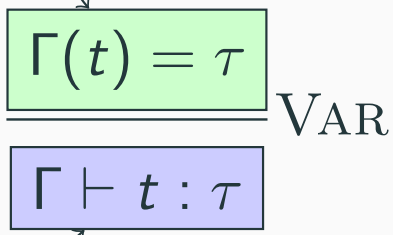
$$\frac{\Gamma(t) = \tau}{\Gamma \vdash t : \tau}\textsc{Var} \qquad \frac{\Gamma \vdash f : \phi \to \alpha \qquad \Gamma \vdash x : \phi}{\Gamma \vdash f\ x : \alpha}\textsc{App}$$

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

- Typvariablen: τ, α, π, ρ
- 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 Γ enthält einen Typ τ für t.“



- Daraus folgt:
- „Variable t hat im Kontext Γ den Typ τ.“

Typisierungsregel für Funktionsanwendungen

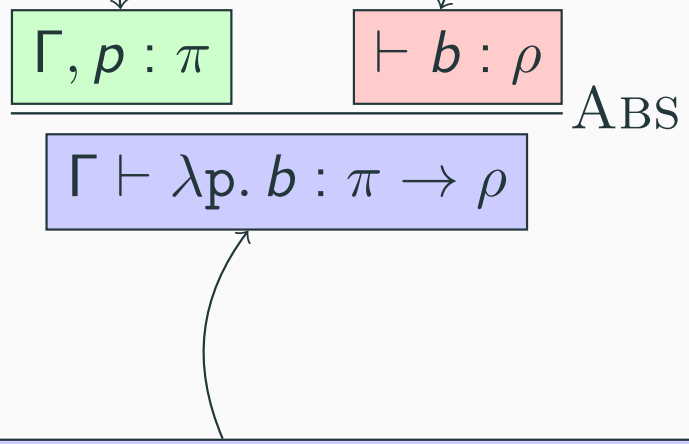
- „f ist im Kontext Γ eine Funktion, die ϕs auf αs abbildet.“
- „x ist im Kontext Γ ein Term des Typs ϕ.“

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

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

Typisierungsregel für Lambdas

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



- Daraus folgt:
- „$\lambda p.\, b$ ist eine Funktion, die πs auf ρs abbildet“

Typinferenz

Vorgehensweise zur Typinferenz:

- Stelle Typherleitungsbaum auf
  - In jedem Schritt werden neue Typvariablen α_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:

$$\begin{aligned} \tau_1 \to \tau_2 &\equiv \texttt{func}(\tau_1, \tau_2) \\ [\tau] &\equiv \texttt{list}(\tau) \end{aligned}$$

etc.

Typinferenz: Übungsaufgaben

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

- „Finde den allgemeinsten Typen α_1 von $\lambda \mathrm{x}.\, f\ x$“

Erinnerung:

- Baum mit durchnummerierten α_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:
$\{\alpha_j = \alpha_k \to \alpha_i\}$

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

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

- Constraint-System auflösen

Typinferenz: Übungsaufgaben

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

- „Finde den allgemeinsten Typen α_1 von $\lambda \mathtt{f}.\, \lambda \mathtt{x}.\, (f\ x)\ x$“

Erinnerung:

- Baum mit durchnummerierten α_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:
$\{\alpha_j = \alpha_k \to \alpha_i\}$

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

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.`

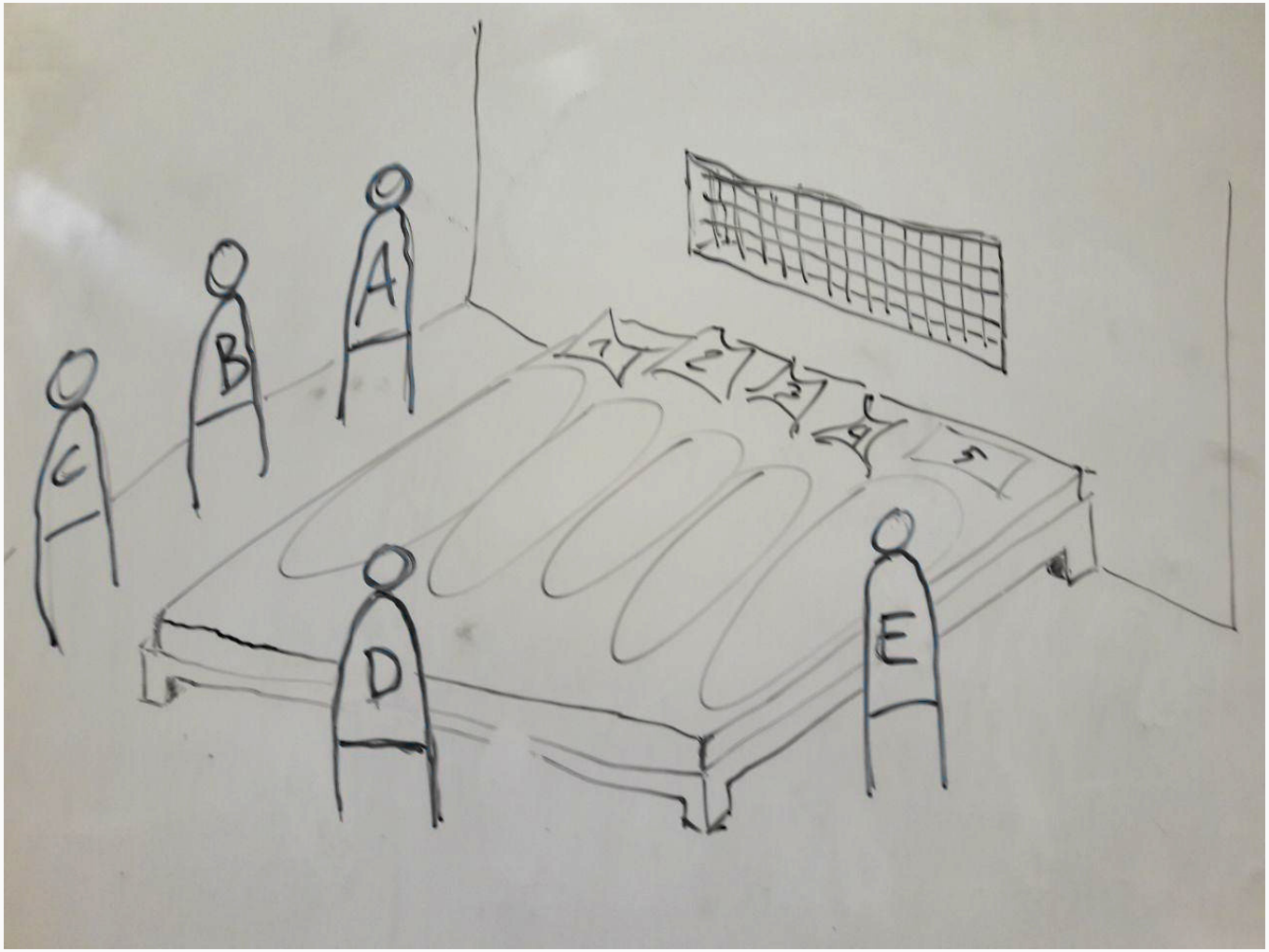
Prolog — Regelsysteme als Programmiersprache

```
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

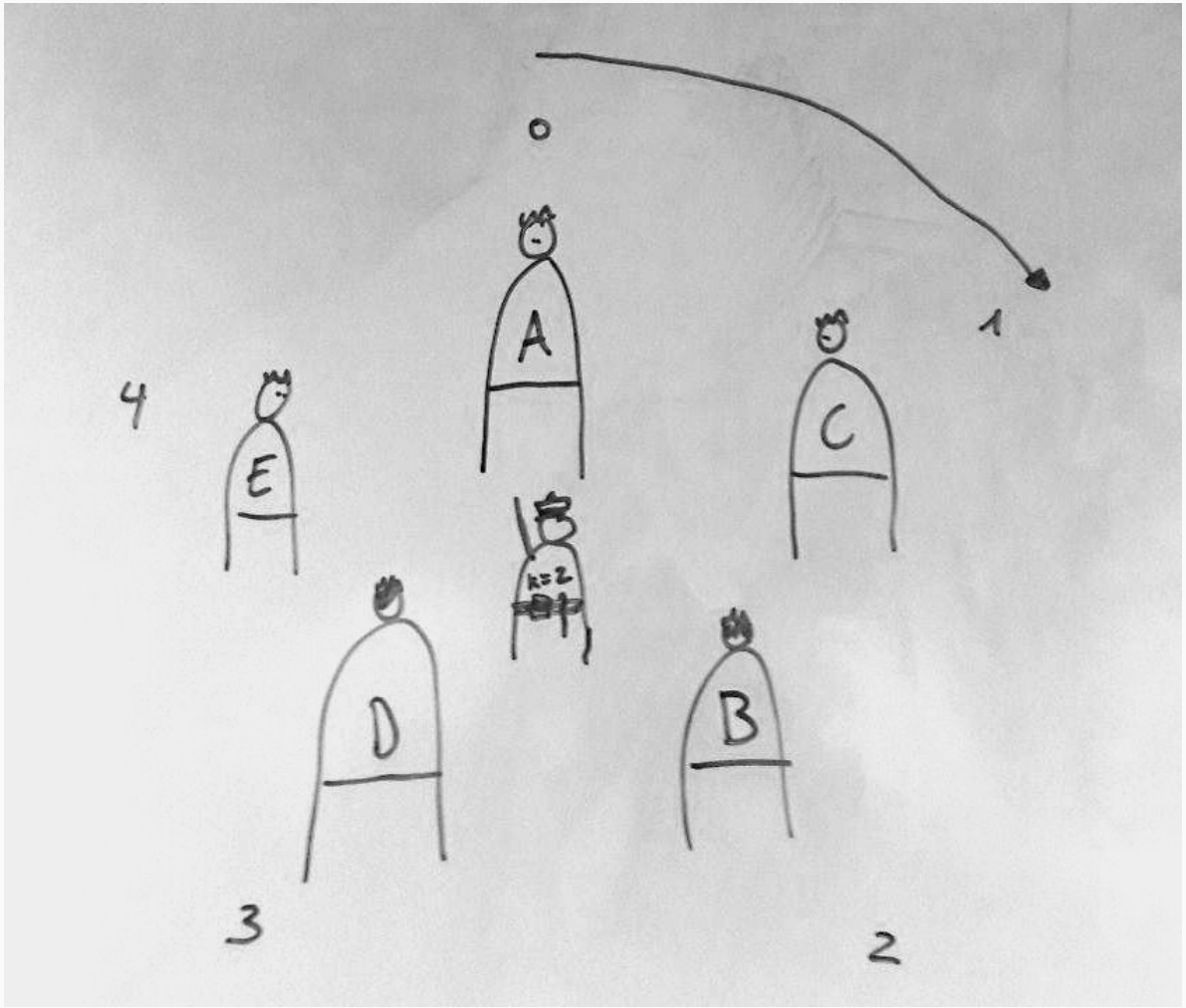
```
% schlafplaetze.pl

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) :-
  helper(L, K, K, X).

helper([X], _C, _K, X) :- !.
helper([H|T], 1, K, X) :- ...
...
```

- Weitere Fälle für `helper/4`:
  - `C = 1` $\leadsto$ 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

Nächste Woche (14. Dezember 2021)

- Blätter 5-7
- Mehr...
  - ...Prolog?
  - ...Typisierung?
  - ...Haskell?