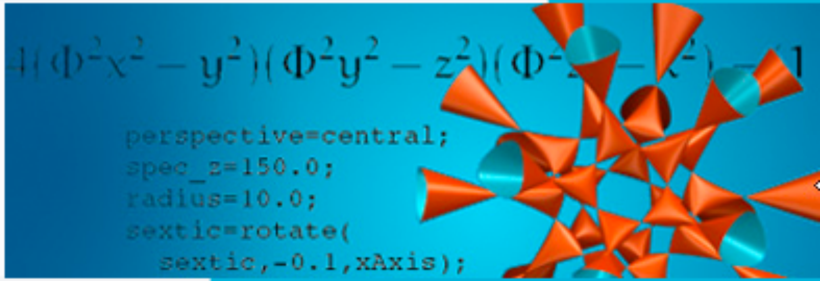
Philipps Universität Marburg

Deklarative Programmierung

Sommersemester 2018

Prof. Christoph Bockisch
(Programmiersprachen und –werkzeuge)
Steffen Dick, Alexander Bille, Johannes Frankenau,
Patrick Frömel, Niclas Schmidt, Jonas Stettin,
Robert Tran, Julian Velten



[The art of Prolog: 1.4 – 2.3, 3 – 3.2]

Substitution

- **Definition**: Eine Substitution ist eine endliche (evtl. leere) Menge von Paaren:
  - $X_i = t_i$
  - X_i ist eine Variable
  - t_i ist ein Term
  - Alle X_i sind verschieden:
    - Für alle i und j mit $i \neq j$ gilt $X_i \neq X_j$
  - Keine Variable, die ersetzt wird, kommt in einem der Terme vor, durch die ersetzt wird:
    - Für alle i und j gilt X_i kommt nicht in t_j vor

- Wir schreiben für eine Substitution typischerweise: θ
- Anwenden der Substitution θ auf den Term A ergibt $A\theta$

Substitution

- Beispiel:
  - $\theta = \{X / isaac\}$
  - A = father(abraham,X).
  - Dann ist $A\theta$ = father(abraham,isaac).

Instanz

- **Definition**: Ein Term *A* ist eine **Instanz eines Terms** *B*,
  - Wenn es eine Substitution *θ* gibt, sodass
  - *A* durch Substitution aus *B* hervorgeht: $A = B\theta$

- Beispiel
  - father(abraham,isaac).
    - Ist eine Instanz von father(abraham,X).
    - Mit der Substitution {X / isaac}
  - mother (sarah, isaac).
    - Ist eine Instanz von mother(X, Y).
    - Mit der Substitution {X / sarah, Y / isaac}

Existenz-Abfragen

- Variablen in Abfragen sind “**existentiell quantifiziert**“:
  - Die Abfrage bedeutet: existieren Terme, sodass die Abfrage erfüllt ist, wenn jede Variable durch einen dieser Terme ersetzt wird?

- Allgemein
  - $(T_1, T_2, \ldots, T_n)$.
  - Abfrage ?- $p(T_1, T_2, \ldots, T_n)$. mit den Variablen $X_1, X_2, \ldots, X_k$
    - Gibt es $x_1, x_2, \ldots, x_k$ sodass $p(T_1, T_2, \ldots, T_n)$.?

- Beispiel:
  - ?- father(abraham, X).
    - “Existiert ein X, sodass abraham der Vater von X ist?“

Existenz-Abfragen: Generalisierung

- **Deduktionsregel**: ***Generalisierung***
  - Eine Existenz-Abfrage ist die logische Konsequenz ihrer Instanzen
  - Gegeben eine Abfrage *P*
  - *P* ist erfüllt, wenn es eine Instanz von *P* mit beliebiger Substitution θ gibt
  - Und $P\theta$ erfüllt ist

- Beispiel
  - Der Fakt father(abraham,isaac).
  - Impliziert, dass es ein X gibt,
  - Sodass ?- father(abraham, X). erfüllt ist, nämlich für θ={X / isaac}

Existenz-Abfragen: Generalisierung

- Bedeutung eines **nicht-Grundterms** (**Generalisierung**)
  - Gegeben eine Abfrage mit logischen Variablen und ein Programm von Fakten
  - Suche einen Fakt, der eine Instanz von der Abfrage ist
    - Existiert so ein Fakt, dann ist diese Instanz die Lösung
    - Repräsentation der Lösung durch Substitution die zu der Instanz führt
    - Gibt es keinen entsprechenden Fakt, ist das Ergebnis Nein
- Im Allgemeinen hat eine Existenz-Abfrage mehrere Lösungen
- Beispiel
  - ?- father(haran,X). hat die Lösungen {X / lot}, {X / milcah}, {X / yiscah}
  - ?- plus (X, Y, 4). hat die Lösungen {X / 0, Y / 4}, {X / 1,Y / 3}, ...

Universelle Fakten

- Variablen können auch in Fakten vorkommen
  - Auch hier: Abstraktion durch Wiederverwendung von Gemeinsamkeiten
  - Variablen fassen mehrere Fakten zusammen

- Beispiel:
  - Anstatt
    - likes(abraham,pomegranates).
    - likes(sarah,pomegranates).
  - Universeller Fakt
    - likes(X, pomegranates).

- Zusammenfassung von Regeln
- Jede Zahl mit 0 multipliziert ergibt 0: times(0, X, 0).

Universelle Fakten

- Variablen in Fakten sind “**universell quantifiziert**“
- Allgemein
  - Ein Fakt $p(T_1, ..., T_n)$. mit den Variablen $X_1, \ldots, X_k$ bedeutet, dass der p für alle Werte der Variablen $X_1, \ldots, X_k$ gilt.
- Beispiel
  - likes(X, pomegranates).
    - ”Für alle X gilt: X mag Granatäpfel.”

Einschränkungen über Variablen

- Variablennamen können mehrfach verwendet werden
- Vorkommen von Variablen müssen **konsistent** durch dieselben Objekte ersetzt werden
- Beispiel
  - Abfragen
    - `?- plus(X, X, 4).`
      - Existiert ein X, sodass X + X gleich 4? – Ja mit {X / 2}
  - Fakten
    - `plus(0,X,X).`
      - Für alle Werte von X ist 0 + X gleich X

Universelle Fakten

- Bedeutung eines **Grundterms** (**Instantiierung**)
  - Gegeben eine Grund-Abfrage und ein Programm von universell quantifizierten Fakten
  - Suche einen Fakt von dem die Abfrage eine Instanz ist

- Beispiel
  - ?- plus(0, 2, 2).
    - Ja, denn es ist eine Instanz des Fakts plus(0,X,X).

Gemeinsame Instanzen

- *C* ist eine **gemeinsame Instanz** von *A* und *B*
  - Wenn *C* eine Instanz von *A*
  - Und eine Instanz von *B* ist

- Das heißt, es gibt θ_1 und θ_2, sodass $C = A\theta_1$ und $C = B\theta_2$

- Beispiel
  - Die Ziele *A:* plus(0,3,Y). und *B*: plus(0,X,X). haben eine gemeinsame Instanz *C*: plus(0, 3, 3).
    - θ_1 = {Y=3}
    - θ_2 = {X=3}

Gemeinsame Instanzen

- Bedeutung eines **nicht-Grundterms**
  - Gegeben eine nicht-Grundabfrage und ein Programm von **universell quantifizierten Fakten**
  - Suche einen Grund-Fakt, der eine **gemeinsame Instanz** von der Abfrage und dem Fakt ist
  - Gibt es eine gemeinsame Instanz,
    - Dann ist die Antwort Ja und die Substitution, die von der Abfrage zur gemeinsamen Instanz führt, ist die Lösung
  - Gibt es keine gemeinsame Instanz,
    - Dann ist die Antwort Nein

- Beantworten einer Existenz-Abfrage mit einem universellen Fakt:
  1. Die gemeinsame Instanz wird durch **Instantiierung** aus dem Fakt hergeleitet
  2. Aus der gemeinsamen Instanz wird durch **Generalisierung** die Abfrage hergeleitet

Konjunktive Abfragen

- Konjunktive Abfragen reihen Ziele aneinander
- Bedeutung des Programms: Sind alle Ziele erfüllt?

- Allgemein
  - *?- Q_1, …, Q_n.*
  - Das Komma (,) steht für die Konjunktion, also ein logisches UND
  - Das Komma ist nicht zu verwechseln mit dem Komma in der Aufzählung von Argumenten
  - Einfache Abfragen sind ein Sonderfall: Konjunktion eines einzelnen Ziels

Konjunktive Abfragen

- Wenn alle Ziele **Grundterme** sind
  - Beantwortung: ist jedes Ziel der Abfrage durch Programm impliziert?
  - Beispiel
    - ?- father(abraham,isaac), male(lot).
      - Ja
- Wenn als Ziele **nicht-Grundterme** vorkommen
  - Gibt es Instanzen der Ziele, die jeweils durch das Programm impliziert werden, wobei gleiche Variablen stets gleich substituiert werden müssen
  - ?- father(terach,X), father(X,Y).
    - Ja {X / abraham, Y / isaac}

Konjunktive Abfragen

- Geteilte Variablen
  - Variablen, die in mehreren Zielen der konjunktiven Abfrage vorkommen

- Das Scope einer logischen Variablen ist der gesamte Ausdruck von konjunktiven Abfragen

- Allgemein:
  - Gegeben ein Ausdruck ?- *p*(*X*),*q*(*X*).
  - Bedeutung: “Gibt es ein X, sodass sowohl ?- p(X). als auch ?- q(X). erfüllt sind?“

Geteilte Variablen

- Geteilte Variablen schränken eine Abfrage ein

- Beispiel:
  - ?- father(haran,X), male(X).
  - Lösungen der Abfrage ?- father(haran,X). werden eingeschränkt auf männliche Kinder
  - Alternativ: Lösungen der Abfrage ?- male(X). werden eingeschränkt auf Objekte deren Vater haran ist

Geteilte Variablen

- Beispiel:
  - ?- father(terach,X), father(X, Y).
  - Söhne von terach werden eingeschränkt auf Objekte, die selbst Vater sind
  - Alternativ: Bestimmen der Objekte Y deren Vater ein Sohn von terach sind
    - Lösung: die Enkel von terach
    - {X / abraham, Y / isaac}, {X / haran, Y / lot}.

Konjunktive Abfragen

- Gegeben eine **konjunktive Abfrage** und ein Programm P
  - Die Abfrage ist eine logische Konsequenz aus P,
  - Wenn alle Ziele in der Konjugation eine logische Konsequenz aus P sind
  - Geteilte Variablen müssen in allen Zielen durch dieselben Terme substituiert werden

- **Bedeutung von konjunktiven Abfragen**
  - Gegeben eine Abfrage A_1, A_2, ... , A_n. und ein Programm P
  - Finde eine Substitution θ, sodass $A_1\theta$ und $A_2\theta$ und ... und $A_n\theta$ Grundinstanzen von Termen in P sind
  - Anwenden derselben Substitution in allen Zielen stellt sicher, dass Variablen konsistent substituiert werden

Regeln

- **Definieren neuer Relationen basierend auf existierenden**

- Allgemein
  - A :- B_1, ..., B_n.
  - A ist der **Regel-Kopf**
  - B_1, ..., B_n ist der **Regel-Körper**

- Beispiel
  - son(X,Y) :- father(Y,X), male(X).
  - daughter(X,Y) :- father(Y,X), female(X).
  - grandfather(X,Y) :- father(X,Z), father(Z,Y).

Regeln: Prozedurale Sicht

- In einer Abfrage, die eine Regel verwendet, wird der Regel-Aufruf durch den Regel-Körper ersetzt

- Beispiel:
  - grandfather(X,Y) :- father(X,Z), father(Z,Y).
    - “Um die Frage zu beantworten, ob X der Großvater von Y ist, beantworte die konjunktive Abfrage ob X der Vater von Z und Z der Vater von Y ist.“
  - ?- grandfather (X, isaac).

  ⇒

  - ?- father(X,Z), father(Z,isaac).

Regeln: Deklarative Sicht

- Bei der Definition einer Regel wird :- verwendet
  - Steht für ← oder “**logische Implikation**“

- Beispiel
  - grandfather(X,Y) :- father(X,Z), father(Z,Y).
  - “Für alle X, Y und Z gilt X ist der Großvater von Y, wenn X der Vater von Z und Z der Vater von Y ist.“

- **Formal**: alle Variablen in einer Regel sind universell quantifiziert
- **Intuitiv**: wir können auch sagen, dass Variablen, die nur im Körper vorkommen existenziell quantifiziert sind
  - ”Für alle X und Y gilt X ist der Großvater von Y, wenn ein Z existiert, sodass X der Vater von Z und Z der Vater von Y ist."

Philipps Universität Marburg

Horn-Klausel

- "**Horn-Klausel**"
  - A :- B_1, ..., B_n.
  - Allgemeine Schreibweise für:
    - Regeln (n > 0)
    - Fakten (n = 0)
    - Abfragen (A ist das Ergebnis, n > 0)
  - Variablen sind universell quantifiziert
  - Scope von Variablen ist die gesamte Horn-Klausel

Modus Ponens

- Gesetz des **universellen Modus Ponens**
  - Informell: „Wenn ein Prädikat P ein Prädikat Q impliziert und P ist wahr, dann ist auch Q wahr.“
  - **Gegeben**
    - Eine Regel *R* mit A :- $B_1, ..., B_n$
    - Die Fakten
      - B_1‘.
      - ...
      - B_n‘.
    - Dann: A‘ kann hergeleitet werden, wenn
      - A‘ :- B_1‘, ..., B_n‘ eine Instanz von R ist
  - Allgemeine Deduktionsregel, schließt ein:
    - Identität (Abfrage und Fakt)
    - Instantiierung (Grundabfrage, universell quantifizierte Fakten)
      - „Suche einen Fakt von dem die Abfrage eine Instanz ist“

Logikprogramme

- Ein Logikprogramm ist eine endliche **Menge an Regeln**
- Ein **existenziell quantifiziertes Ziel** ist die Logische Konsequenz eines Programms,
  - Wenn es eine Klausel P mit der Grundinstanz A :- B_1, ..., B_n (n >= 0) gibt,
  - Sodass B_1, ..., B_n jeweils logische Konsequenz von P sind
  - Und A eine Instanz von der Grundabfrage G ist
- G ist eine logische Konsequenz des Programms P genau denn, wenn G durch eine endliche Anzahl von Anwendungen des Modus Ponens hergeleitet werden kann

Logikprogramme

Universeller Modus Ponens impliziert die Abfrage mit Lösung {S/lot}

- Beispiel
  - son(X,Y) :- father(Y,X), male(X).
  - ? son(S, haran).

Instanz

Substitution {S/lot}

Fakten in unserem Programm

  - son(lot, haran) :- father(haran, lot), male(lot).

Grundinstanz

Substitution {X/lot, Y/haran}

  - son(X,Y) :- father(Y,X), male(X).

Logikprogramme

- Für eine Relation können mehrere Regeln angegeben werden
  - Regeln bilden Alternativen (Disjunktion)
- Die Menge der Regeln für dieselbe Relation (dasselbe Prädikat) wird “Prozedur“ genannt
- Beispiel
  - son(X,Y) :- father(Y,X), male(X).
  - son(X,Y) :- mother(Y,X), male(X).

Hilfs-Relationen

- Vereinfachung von Relationen durch Hilfs-Relationen
  - Vermeidung von Redundanz/Vereinfachung von Aufzählungen

- Beispiel
  - Umständlich
    - grandparent(X,Y) :- father(X,Z), father(Z,Y).
    - grandparent(X,Y) :- father(X,Z), mother(Z,Y).
    - grandparent(X,Y) :- mother(X,Z), father(Z,Y).
    - grandparent(X,Y) :- mother(X,Z), mother(Z,Y).
  - Einfacher
    - parent (X, Y) :- father (X, Y).
    - parent(X,Y) :- mother(X,Y).
    - grandparent(X,Y) :- parent(X,Z), parent(Z,Y).

Bedeutung von Prolog

- Beschreibung der Semantik von Prolog durch abstrakten Interpreter
  - Gegeben ein Programm und eine Abfrage
  - Antwort:
    - Ja: Das Programm impliziert die Abfrage
    - Nein: Das Programm impliziert die Abfrage nicht
    - Keine Antwort: Interpreter terminiert nicht, wenn die Abfrage nicht in einer endlichen Anzahl Schritte herleitbar ist

Abstrakter Interpreter

- **Grundidee**
  - **Resolvente**: Ziel, welches im aktuellen Schritt hergeleitet werden soll
    - Im Allgemeinen: Konjunktive Abfrage
  - **Reduktion**: Berechnung der nächsten Resolvente durch Anwendung des Moduls Ponens
    - Reduktion eines Ziels *G* unter einem Programm *P*:
    - Ersetze *G* durch den Körper einer Instanz einer Regel in *P*
    - Wobei der Kopf der Regel identisch ist zu *G*
  - **Trace**: Sequenz der während der Auswertung berechneten Resolventen
- Hier: **Einschränkung auf Grund-Abfragen**

Abstrakter Interpreter

- Gegeben eine Grund-Abfrage *G* und ein Programm *P*
- **Algorithmus**
  - Initialisiere die Resolvente zu G
  - Solange die Resolvente nicht leer
    - Wähle ein Ziel A von der Resolvente
    - Wähle eine Grundinstanz einer Klausel $A' \text{ :- } B_1, \ldots, B_n$ aus P
      - Sodass *A* und *A‘* identisch sind
      - Wenn keine solche Instanz existiert, beende den Algorithmus
    - Ersetze A durch $B_1, \ldots, B_n$
    - Ergebnis
      - Ja, wenn die Resolvente leer ist
      - Nein, sonst

Abstrakter Interpreter

- Programm:
  - father(abraham,isaac).
  - father(haran,lot).
  - father(haran,milcah).
  - father(haran,yiscah).
  - male(isaac).
  - male(lot).
  - female(milcah).
  - female(yiscah).
  - son(X, Y) :- father (Y, X), male (X) .
  - daughter(X, Y) :- father (Y, X), female (X).
- G: ?- son(lot,haran).
- Interpretation:
  - Initialisierung
    - Resolvente: son(lot,haran)
  - Resolvente ist nicht leer

Abstrakter Interpreter

- Programm:
  - father(abraham,isaac).
  - father(haran,lot).
  - father(haran,milcah).
  - father(haran,yiscah).
  - male(isaac).
  - male(lot).
  - female(milcah).
  - female(yiscah).
  - son(X, Y) :- father (Y, X), male (X) .
  - daughter(X, Y) :- father (Y, X), female (X).
- G: ?- son(lot,haran).
- Interpretation:
  - Resolvente: son(lot,haran)
  - Wähle Ziel aus der Resolvente
    - son(lot,haran)
  - Wähle Grundinstanz einer Regel aus P
    - son(lot,haran) :- father(haran,lot), male(lot)
  - Ersetze Ziel der Resolvente durch Regel-Körper
    - Neue Resolvente: father(haran,lot), male(lot)
  - Resolvente ist nicht leer

Einziges Ziel

“Raten“ einer passenden Regel

Abstrakter Interpreter

- Programm:
  - father(abraham,isaac).
  - father(haran,lot).
  - father(haran,milcah).
  - father(haran,yiscah).
  - male(isaac).
  - male(lot).
  - female(milcah).
  - female(yiscah).
  - son(X, Y) :- father (Y, X), male (X) .
  - daughter(X, Y) :- father (Y, X), female (X).
- G: ?- son(lot,haran).
- Interpretation:
  - Resolvente: father(haran,lot), male(lot)
  - Wähle Ziel aus der Resolvente
    - father(haran,lot)
  - Wähle Grundinstanz einer Regel aus P
    - father(haran,lot)
  - Ersetze Ziel der Resolvente durch Regel-Körper
    - Neue Resolvente: male(lot)
  - Resolvente ist nicht leer

Fakt: Regel-Körper ist leer

Abstrakter Interpreter

- Programm:
  - father(abraham,isaac).
  - father(haran,lot).
  - father(haran,milcah).
  - father(haran,yiscah).
  - male(isaac).
  - male(lot).
  - female(milcah).
  - female(yiscah).
  - son(X, Y) :- father (Y, X), male (X) .
  - daughter(X, Y) :- father (Y, X), female (X).
- G: ?- son(lot,haran).
- Interpretation:
  - Resolvente: male(lot)
  - Wähle Ziel aus der Resolvente
    - male(lot)
  - Wähle Grundinstanz einer Regel aus P
    - male(lot)
  - Ersetze Ziel der Resolvente durch Regel-Körper
    - Neue Resolvente:
  - Resolvente ist leer

Ergebnis: Ja

Abstrakter Interpreter

- Wahlen
  - Auswahl des Ziels aus der Resolvente ist beliebig
  - Auswahl der Regel ist schwierig
    - Im Allgemeinen müssen alle Regeln und alle Grund-Fakten ausprobiert werden
    - Es können mehrere Regeln in Frage kommen
- Reale Prolog Implementierungen müssen die Auswahl der Regel nicht raten
  - Liegt außerhalb des Scope der Veranstaltung

Vergleiche

- Vergleichen von Termen in Abfragen
  - Variablen werden durch Terme substituiert
  - Ohne Vergleich: Ausprobieren aller möglicher Terme
  - Mit Vergleich: Einschränkung der probierten Terme

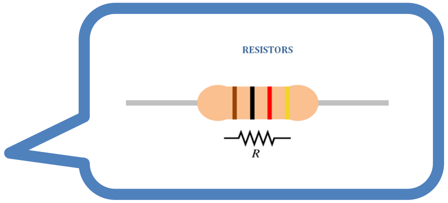
- Beispiel
  - `brother(Brother,Sib) :- parent(Parent,Brother), parent(Parent,Sib), male(Brother).`
  - `?- brother(isaac, isaac).` ergibt Ja
    - Aber das entspricht nicht der Intuition hinter der Relation brother
  - Einschränkung Brother und Sib müssen unterschiedliche Objekte sein
    - `brother(Brother,Sib) :- parent(Parent,Brother), parent(Parent,Sib), male(Brother), Brother /== Sib.`

Einschränkung durch Vergleich

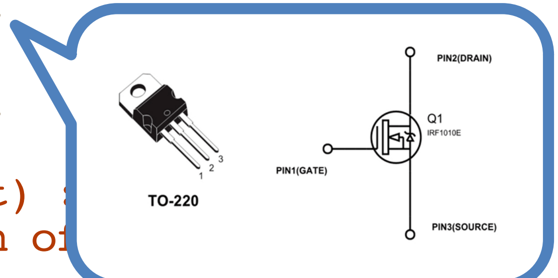
Kommentare zur Beschreibung von Relationen

```
resistor(power,n1).
resistor(power,n2).
transistor(n2,ground,n1).
transistor(n3,n4,n2).
transistor(n5,ground,n4).

/*inverter (Input, Output) :-
  Output is the inversion of Input.*/
inverter(Input,Output) :- transistor(Input,ground,Output),
resistor(power,Output).

/*nand_gate(Input1,Input2,Output) :-
  Output is the logical nand of Input 1 and Input2.*/
nand_gate(Input1, Input2, Output) :- transistor(Input1,X,Output),
transistor(Input2,ground,X),resistor(power,Output).

/*and_gate (Input1,Input2,Output) :-
  Output is the logical and of Input1 and Input2.*/
and_gate(Input1, Input2, Output) :- nand_gate(Input1,Input2,X),
inverter(X,Output).
```

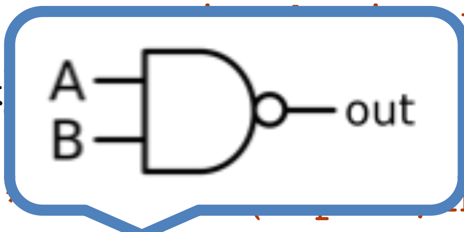
Kommentare zur Beschreibung von Relationen

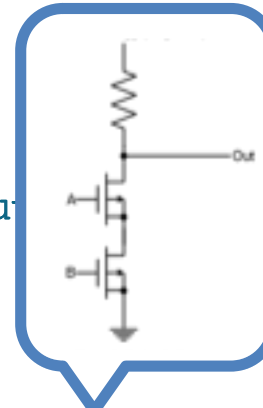



```
resistor(power,n1).
resistor(power,n2).
transistor(n2,ground,n1).
transistor(n3,n4,n2).
transistor(n5,ground,n4).
```



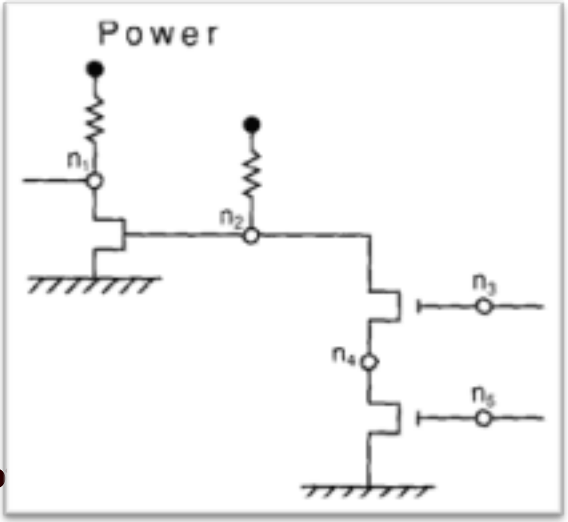

```
/*inverter (Input, Output) :
                  rsion of
i            t) :- transistor(Input,ground,Outpu
r            t).
```






```
/*            nput2,Output) :-
  Output is the logical nand of Input 1 and Input2.*/
nand_gate(Input1, Input2, Output) :- transistor(Input1,X,Output),
transistor(Input2,ground,X),resistor(power,Output).

/*and_gate (Input1,Input2,Output) :-
  Output is the logical and of Input1 and Input2.*/
and_gate(Input1, Input2, Output) :- nand_gate(Input1,Input2,X),
inverter(X,Output).
```

Kommentare zur Beschreibung von Relationen




```
resistor(power,n1).
resistor(power,n2).
transistor(n2,ground,n1).
transistor(n3,n4,n2).
transistor(n5,ground,n4).

/*inverter (Input, Output) :-
  Output is the inversion of Input.*/
inverter(Input,Output) :- transistor(Input,gro
resistor(power,Output).

/*nand_gate(Input1,Input2,Output) :-
  Output is the logical nand of Input 1 and Input2.*/
nand_gate(Input1, Input2, Output) :- transistor(Input1,X,Output),
transistor(Input2,ground,X),resistor(power,Output).

/*and_gate (Input1,Input2,Output) :-
  Output is the logical and of Input1 and Input2.*/
and_gate(Input1, Input2, Output) :- nand_gate(Input1,Input2,X),
inverter(X,Output).
```

Kommentare zur Beschreibung von Relationen

Für jede Prozedur:
Schema des Prädikats
gefolgt von textueller Beschreibung

```
resistor(power,n1).
resistor(power,n2).
transistor(n2,ground,n1).
transistor(n3,n4,n2).
transistor(n5,ground,n4).

/*inverter (Input, Output) :-
  Output is the inversion of Input.*/
inverter(Input,Output) :- transistor(Input,ground,Output),
resistor(power,Output).

/*nand_gate(Input1,Input2,Output) :-
  Output is the logical nand of Input 1 and Input2.*/
nand_gate(Input1, Input2, Output) :- transistor(Input1,X,Output),
transistor(Input2,ground,X),resistor(power,Output).

/*and_gate (Input1,Input2,Output) :-
  Output is the logical and of Input1 and Input2.*/
and_gate(Input1, Input2, Output) :- nand_gate(Input1,Input2,X),
inverter(X,Output).
```

Kommentare zur Beschreibung von Relationen

```
resistor(power,n1).
resistor(power,n2).
transistor(n2,ground,n1).
transistor(n3,n4,n2).
transistor(n5,ground,n4).

/*inverter (Input, Output) :-
  Output is the inversion of Input.*
inverter(Input,Output) :- transistor
resistor(power,Output).

/*nand_gate(Input1,Input2,Output) :-
  Output is the logical nand of Input 1 and Input2.*/
nand_gate(Input1, Input2, Output) :- transistor(Input1,X,Output),
transistor(Input2,ground,X),resistor(power,Output).

/*and_gate (Input1,Input2,Output) :-
  Output is the logical and of Input1 and Input2.*/
and_gate(Input1, Input2, Output) :- nand_gate(Input1,Input2,X),
inverter(X,Output).
```

?- and_gate(In1,In2,Out).

Ja.

In1 = n3, In2 = n5, Out = n1

Lösung: and_gate(n3,n5,n1)

Datenstrukturen

- Faktdatenbank repräsentiert Datenstruktur nur implizit

- Wir wollen eine sprechende Beschreibung der Daten aufbauen

- Konvention:
  - Hinzufügen eines ersten Arguments zu allen Zielen in der Faktendatenbank
  - Grundfakten: Bezeichner
  - Nicht-Grundfakten: zusammengesetzter Term

Datenstrukturen

```
/*resistor(R,Node1,Node2) :-
  R is a resistor between Node1 and
  Node2.*/
resistor(r1,power,n1).
resistor(r2,power,n2).

/*transistor(T,Gate,Source,Drain) :-
  T is a transistor whose gate is
Gate,
  source is Source, and drain is
Drain.*/
transistor(t1,n2,ground,n1).
transistor(t2,n3,n4,n2).
transistor(t3,n5,ground,n4).

/*inverter(I,Input, Output) :-
  I is an inverter that inverts Input
  to Output.*/
inverter(inv(T,R),Input,Output) :-
  transistor(T,Input,ground,Output),
  resistor(R,power,Dutput).
```

```
/*nand_gate(Nand,Input1,Input2,Output
)
  :- Nand is a gate forming the
logical
  and, Output, of Input1 and
Input2.*/
nand_gate(nand(T1,T2,R),Input1,Input2
,Output) :-
  transistor(T1,Input1,X,Output),
  transistor(T2,Input2,ground,X),
  resistor(R,power,Output).

/*and_gate (And,Input1 ,Input2,
Output) :-
  And is a gate forming the logical
and,
  Output, of Input1 and Input2.*/
and_gate(and(N,I),Input1,Input2,
Output) :-
  nand_gate(N,Input1,Input2,X),
  inverter(I,X,Output).
```

Datenstrukturen

```
/*resistor(R,Node1,No
   R is a resistor
   Node2.*/
resistor(r1,power,n1)
resistor(r2,power,n2)

/*transistor(T,Gate,
   T is a transistor whose gate is
Gate,
   source is Source, and drain is
Drain.*/
transistor(t1,n2,ground,n1).
transistor(t2,n3,n4,n2).
transistor(t3,n5,ground,n4)

/*inverter(I,Input, Output
   I is an inverter that
   to Output.*/
inverter(inv(T,R),Input,Outpu
   transistor(T,Input
   resistor(R,power,Output).
```

```
Input2,Output
ing the
and, Output, of Input1 and

nd(T1,T2,R),Input1,Input2
   transistor(T1,Input1,X,Output),
   transistor(T2,Input2,ground,X),
   resistor(R,power,Output).

/*and_gate (And,Input1 ,Input2,
Output) :-
the logical
Input2.*/
,Input2,
,X),
```

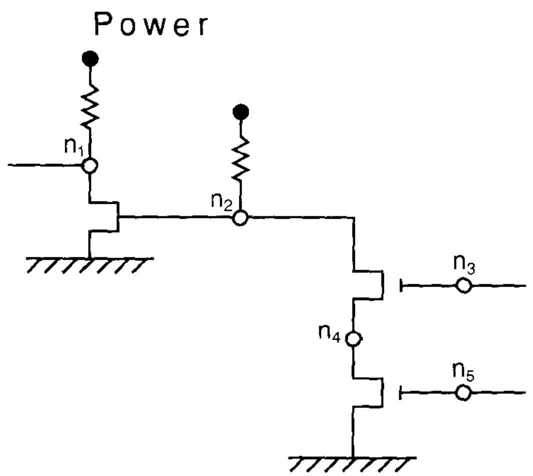
Erstes Argument: Variable, eine sprechende Beschreibung des Ziels

Grundfakt: Bezeichner

Nicht-Grundfakt: Struktur wird in Regelkopf als zusammengesetzter Term vorgegeben

Nicht-Grundfakt: Unterterme werden in Regelkörper gebunden

Datenstrukturen



- Die Abfrage:
  - `?- and_gate(G, In1, In2, Out).`
- Hat die Lösung
  - `{G=and(nand(t2,t3,r2),inv(t1,r1)),In1=n3,In2=n5,Out=n1}`

Datenabstraktion

- Fakten können durch geschachtelte Relationen dargestellt werden
  - Zerlegung in sinnvolle Teilfakten
  - Bessere Abstraktion von Datenrepräsentation

- Beispiel
  - Anstatt
    - `course(complexity, monday, 9, 11, david, harel, feinberg, a).`
  - Mit Datenabstraktion
    - `course(complexity, time(monday, 9, 11), lecturer(david, harel), location(feinberg, a)).`

Zeit, Dozent und Ort auch als separate Fakten

Datenabstraktion

```
lecturer(Lecturer,Course) :-
  course(Course,Time,Lecturer,Location).
duration(Course,Length) :-
  course(Course,time(Day,Start,Finish),Lecturer,Location),
  plus(Start,Length,Finish).

teaches(Lecturer,Day) :-
  course(Course,time(Day,Start,Finish),Lecturer,Location).

?- lecturer(L,complexity).
```

Ja.
L = lecturer(david,harel)

Ich muss nicht wissen, welche Eigenschaften ein Dozent hat.

Rekursive Regeln

- Auch bei Logik-Regeln gibt es Selbstähnlichkeit

- Beispiel

```
grandparent(Ancestor,Descendant) :-
parent(Ancestor,Person), parent(Person,Descendant).

greatgrandparent(Ancestor,Descendant) :-
parent(Ancestor,Person), grandparent(Person,Descendant).

greatgreatgrandparent(Ancestor,Descendant) :-
parent(Ancestor,Person), greatgrandparent(Person, Descendant).
```

Rekursive Regeln

- Mittel um diese Redundanz zu vermeiden: Rekursive Definition
  - ancestor(Ancestor,Descendant) :-
    parent(Ancestor,Person), ancestor(Person,Descendant).

- Nicht-rekursive Regel für Rekursionsabbruch benötigt
- Regeln repräsentieren logische Fakten!
  - Ein Fakt ancestor(X,X). würde zwar zum Rekursionsabbruch führen, aber auch aussagen, dass eine Person ihr eigener Vorfahre ist.

```
/*ancestor (Ancestor,Descendant) :-
  Ancestor is an ancestor of  Descendant.*/
ancestor(Ancestor,Descendant) :-
    parent(Ancestor,Descendant).
ancestor(Ancestor,Descendant) :-
    parent(Ancestor,Person), ancestor(Person,Descendant).
```

Rekursive Programmierung: Arithmetik

- Die Zahl 0 ist eine natürliche Zahl
- Der Nachfolger von einer natürlichen Zahl ist eine natürliche Zahl

```
/*natural_ number (X) :-
  X is a natural number.*/
natural_number(0).
natural_number(s(X)) :- natural_number(X).
```

- Natürliche Zahlen können repräsentiert werden als
  - 0, s(0), s(s(0)), etc.
  - Abkürzend $s^n(0)$ für n Anwendungen der Regel S(X) auf 0

Arithmetik

```
/* X ’<=’ Y :-
   X and Y are natural numbers,
   such that X is less than or equal to Y.*/
'<='(0,X) :- natural_number(X).
'<='(s(X),s(Y)) :- '<='(X,Y).

/* plus(X,Y,Z) :-
   X, Y, and Z are natural numbers,
    such that Z is the sum of X and Y.*/
plus(0,X,X) :- natural_number(X).
plus(s(X),Y,s(Z)) :- plus(X,Y,Z).
```

Listen

- Listen als rekursive Datenstruktur
  - Erstes Argument ist ein Element der Liste
  - Zweites Argument ist der Rest der Liste
  - Konstantes Symbol für Abbruch der Liste: `[ ]`

- Schreibweisen
  - cons-Zelle
    - `[a|[ ]]`
    - `[a|[b|[ ]]]`
    - `[a|[b|[c|[ ]]]]`
    - `[a|X]`
    - `[a|[b|X]]`
  - Element-Syntax
    - `[a]`
    - `[a,b]`
    - `[a,b,c]`
    - `[a|X]`
    - `[a,b|X]`

Definition von Liste

```
/* list(Xs) :- Xs is a list. */
list([]).
list([X | Xs]) :- list (Xs).
```

Prozeduren über Listen

```
/*member (Element,List) :-
   Element is an element of the list List.*/
member(X, [X|Xs]).
member(X, [Y|Ys]) :- member(X,Ys).
```

- Die Prozedur member kann auf verschiedene Weise gebraucht werden:
  - Ist ein b in der Liste [a,b,c] enthalten?
    - `?- member(b,[a,b,c]).`
  - Sei X ein Element aus der Liste [a,b,c]
    - `?- member(X,[a,b,c]).`
  - Sei X eine Liste die das Element b enthält
    - `?- member(b,X).`

Prozeduren über Listen

```
/* prefix (Prefix,List) :-
   Prefix is a prefix of List.*/
prefix([ ] , Ys).
prefix([X|Xs],[X|Ys]) :- prefix(Xs,Ys).

/* suffix C Suffix, List) :-
   Suffix is a suffix of List.*/
suffix(Xs,Xs).
suffix(Xs,[Y|Ys]) :- suffix(Xs,Ys).

/* append (Xs, Ys, Zs) :-
   Zs is the result of concatenating the
   lists Xs and Ys.*/
append([], Ys, Ys).
append([X | Xs], Ys, [X | Zs]) :-
   append(Xs, Ys, Zs).
```