Programmiersprachen

Alexandru Berlea

Institut für Informatik
TU München

Wintersemester 2006/2007

Konjunktive Anfragen mit Variablen

- Wenn eine Variable in verschiedenen Zielen in einer konjunktiven Anfrage erscheint, dann bezieht sie sich immer auf das selbe Objekt.
  - Bsp.: `father(haran,X),male(X)`
- Eine konjunktive Anfrage ist die Folge eines Programms, wenn jedes Ziel eine Folge des Programms ist, wobei jede Variable mit dem selben Term in verschiedenen Zielen ersetzt wird.

Regeln

- **Regeln** erlauben, neue Beziehungen mit Hilfe existierender Beziehungen zu definieren.

- Regeln sind Aussagen der Form:

$$A \leftarrow B_1, B_2, \ldots, B_n$$

  - A heißt **Kopf** der Regel.
  - Die Konjunktion von Zielen $B_1, B_2, \ldots, B_n$ heißt **Rumpf** der Regel.
  - Fakten sind Regeln im Spezialfall $n = 0$. Im Allgemeinen ist ein Programm eine endliche Menge von Regeln (statt eine Menge von Fakten).

- Fakten, Anfragen und Regeln heißen auch **(Horn) Klauseln**.

Regeln

- Variablen in Regeln sind (wie in Fakten) universell quantifiziert.
  - `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).`
- "←" stellt logische Implikation dar
  $\Longrightarrow$**Modus ponens** als Deduktionsregel:

  Aus $R = (A \leftarrow B_1, B_2, \ldots, B_n)$ und $B'_1, B'_2, \ldots, B'_n$ folgt A' wenn $A' \leftarrow B'_1, B'_2, \ldots, B'_n$ eine Instanz von $A \leftarrow B_1, B_2, \ldots, B_n$ ist.

  - Identität und Instantiierung sind Spezialfälle des Modus ponens.

Prozedurale vs. logische Interpretation der Regeln

Bsp.: `grandfather(X,Y)` $\leftarrow$ `father(X,Z), father(Z,Y).`

- Prozedurale I.: *Um die Anfrage "ist X der Großvater von Y?" zu beantworten, beantworte die konjunktive Anfrage "ist X der Vater von Z und Z der Vater von Y?".*

- Logische I.: *Für alle X, Y und Z, X ist der Großvater von Y, wenn X der Vater von Z und Z der Vater von Y ist.*

Logische Programme als deduktive Datenbanken

- Außer Relationen enthält eine **deduktive/logische Datenbank** Regeln, die erlauben, neue Relationen aus bestehenden Relationen zu gewinnen.
  $\Longrightarrow$ Logische Programme können als deduktive Datenbanken betrachtet werden.
  - Grundrelationen werden via Fakten spezifiziert.
  - Regeln entsprechen der logischen Regeln.

Beispiele

- Als Grundrelationen nehmen wir `father/2`, `mother/2`, `male/1` und `female/1` an, wobei die Zahl die Stelligkeit der jeweiligen Relation angibt.

- Wir definieren neue Relationen mit Hilfe der bestehenden Relationen via Regeln:
  - `parent(Parent,Child)` ← `father(Parent,Child).`
    `parent(Parent,Child)` ← `mother(Parent,Child).`
    (Mehrere Regeln mit dem selben Kopf definieren eine Disjunktion.)

  - `procreated(Man,Woman)` ←
    `father(Man,Child), mother(Woman,Child).`

Beispiele

```
brother(Brother,Sib) ←
   parent(Parent,Brother),parent(Parent,Sib),male(Brother).
```

- Problem: `brother(X,X)?` gilt für jedes männliche Kind `X`.
- Lösung: Wir nehmen an, es existiert ein vordefiniertes Prädikat `≠(Term1,Term2)`, das wahr ist, wenn `Term1` verschieden von `Term2` ist. Wir schreiben das Prädikat infixiert: `Term1 ≠ Term2`.

$\Longrightarrow$

```
brother(Brother,Sib) ←
   parent(Parent,Brother),parent(Parent,Sib),
   male(Brother),Brother≠Sib.
```

Beispiele

- `uncle(Uncle,Person)` $\leftarrow$
  `brother(Uncle,Parent),parent(Parent,Person)`

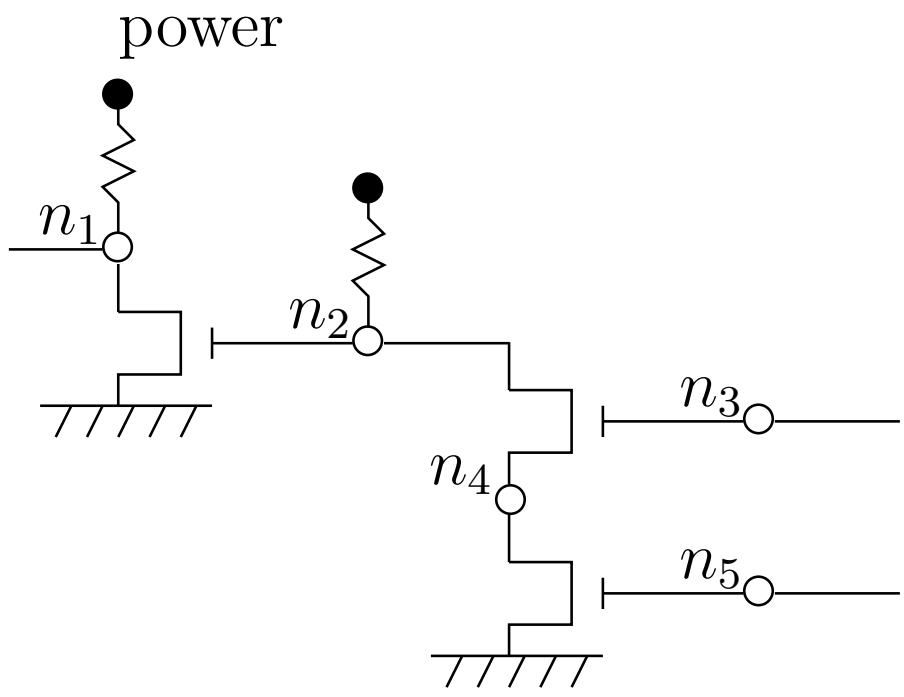
- `sibling(Sib1,Sib2)` $\leftarrow$
  `parent(Parent,Sib1),parent(Parent,Sib2),Sib1`$\neq$`Sib2.`

- `cousin(Cousin1,Cousin2)` $\leftarrow$
  `parent(Parent1,Cousin1),parent(Parent2,Cousin2),`
  `sibling(Parent1,Parent2).`

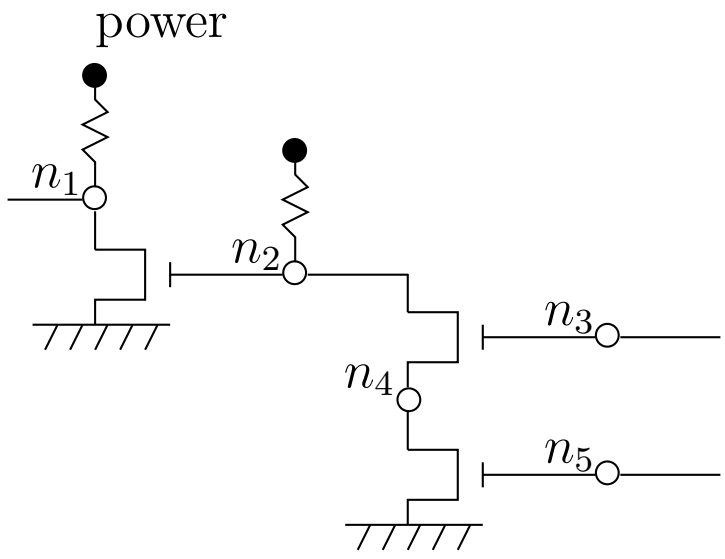
Beispielanfragen

- Sind Hanah und Isaac Geschwister: `sibling(hanah,isaac)?`
- Wer ist der Onkel von Hanah: `uncle(X,hanah)?`
- Welche Geschwisterpaare gibt es: `sibling(X,Y)?`

Beispiel



Beispiel




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

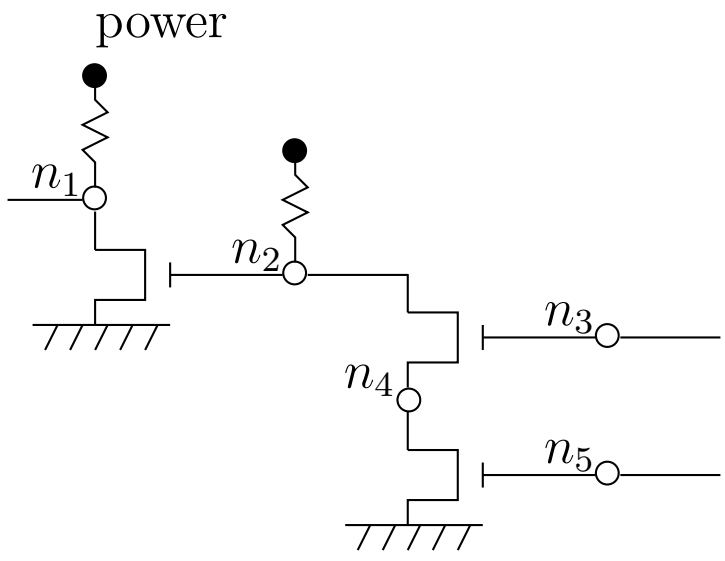
Beispiel

- ```
  not(Input,Output) ←
      transistor(Input,ground,Output),
      resistor(power,Output).
  ```

- ```
  nand(Input1,Input2,Output) ←
      transistor(Input1,X,Output),
      transistor(Input2,ground,X),
      resistor(power,Output).
  ```

- ```
  and(Input1,Input2,Output) ←
      nand(Input1,Input2,X),
      not(X,Output).
  ```
  ```

Beispielanfragen




```
and(In1,In2,Out)?
⟹{In1=n3,In2=n5,Out=n1}
```

Strukturierte Daten und Datenabstraktion

- Flache Darstellung einer Vorlesung:

```
course(compilerbau,montag,9,11,helmut,seidl,MW,1001).
```

  $\longrightarrow$ Kompakte aber unübersichtliche Darstellung

- Strukturierte D.:

```
course( compilerbau,
        zeit(montag,9,11),
        lecturer(helmut,seidl),
        room(MW,1001)).
```

  $\longrightarrow$ Abstraktion der für die jeweiligen Ziele unwesentlichen Details, z.B.:

```
lecturer(Lecturer,Course) ← course(Course,Time,Lecturer,Loc).
teaches(Lecturer,Day)← course(Course,zeit(Day,S,F),Lecturer,Loc).
```

Rekursive Regeln

```
ancestor(Ancestor,Descendant) ←
     parent(Ancestor,Descendant)

ancestor(Ancestor,Descendant) ←
     parent(Ancestor,Person), ancestor(Person,Descendant)
```

Logische Programme zur Bearbeitung rekursiver Datenstrukturen

Bäume:

- ```
  bintree(void).
  bintree(tree(Element,Left,Right)) ←
     bintree(Left), bintree(Right).
  ```

- ```
  member(X,tree(X,Left,Right)).
  member(X,tree(Y,Left,Right)) ← member(X,Left).
  member(X,tree(Y,Left,Right)) ← member(X,Right).
  ```

Bsp.: Baumisomorphie

- Zwei Bäume t_1 und t_2 sind isomorph, wenn t_2 durch eine Umordnung der Zweige in t_1 erhalten werden kann.
- Bäume in denen die Reihenfolge der Söhne unwichtig ist = ungeordnete Bäume

```
isotree(void,void).

isotree(tree(X,Left1,Right1),tree(X,Left2,Right2)) ←
  isotree(Left1,Left2), isotree(Right1,Right2).

isotree(tree(X,Left1,Right1),tree(X,Left2,Right2)) ←
  isotree(Left1,Right2), isotree(Right1,Left2).
```

Bsp.: Substitution in Bäumen

- Problem: Ersetze im Baum T_1 alle Vorkommen von X durch Y.

- Lösung: Die Eingabedaten und der Ergebnisbaum T_2 sind Argumente eines Prädikats, `substitute(X,Y,`T_1`,`T_2`)`, definiert wie folgt:

```
substitute(X,Y,void,void).
substitute(X,Y,tree(Info,Left,Right),
           tree(Info1,Left1,Right1)) ←
  replace(X,Y,Info,Info1),
  substitute(X,Y,Left,Left1),
  substitute(X,Y,Right,Right1).

replace(X,Y,X,Y).
replace(X,Y,Z,Z) ← X ≠ Z.
```

Listen

- Listen sind ein Spezialfall von Binärbäumen und können syntaktisch definiert werden mit zwei Konstrukten:
  - **Leere Liste**: `[]`
  - **Nicht-leere Liste**: : `[X|Y]`
    - `X` = das erste Element
    - `Y` = der Rest der Liste.
- Syntaktische Konvention: `[a|[b|[c|[]]]]` $\equiv$ `[a,b,c]`

Bsp.: member

```
member(X,[X|Xs]).
member(X,[Y|Ys]) ← member(X,Ys).
```

Bsp.: prefix, suffix, suli

```
prefix([],Ys).
prefix([X|Xs],[X|Ys]) ← prefix(Xs,Ys).

suffix(Xs,Xs).
suffix(Xs,[Y|Ys]) ← suffix(Xs,Ys).

suli(Xs,Ys) ← prefix(Ps,Ys), suffix(Xs,Ps).
```

Bsp.: append

```
append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]) ← append(Xs,Ys,Zs).
```

- `append([a,b,c],[d,e],Xs)?` ergibt *Xs=[a,b,c,d,e]*.
- `append(Xs,[d,e],[a,b,c,d,e])?` ergibt *Xs=[a,b]*.
- `append(As,Bs,[a,b,c,d])?` ergibt die verschiedenen möglichen Aufspaltungen der Liste `[a,b,c,d]`.
  $\longrightarrow$ `append` kann benutzt werden, um Listen aufzuspalten...

Bsp.: append zum Listenaufspalten

```
prefix(Xs,Ys) ← append(Xs,As,Ys).

suffix(Xs,Ys) ← append(As,Xs,Ys).

member(X,Ys) ← append(As,[X|Xs],Ys).

last(X,Xs) ← append(As,[X],Xs).
```

Bsp.: Listen Umdrehen

- Eine mögliche Lösung:

```
reverse([],[]).
reverse([X|Xs],Zs)← reverse(Xs,Ys),append(Ys,[X],Zs).
```

- Andere Lösung:

```
reverse(Xs,Ys) ← reverse(Xs,[],Ys).
reverse([X|Xs],Acc,Ys) ← reverse(Xs,[X|Acc],Ys).
reverse([],Ys,Ys).
```

- Um die zwei Lösungen bezüglich der Effizienz vergleichen zu können, müssen wir den zugrunde liegenden Berechnungsmodell kennen.
  (☞ Nächster Abschnitt: Berechnungsmodell der logischen Programme)
  Im Bsp., Anzahl von Deduktionschritten:
  - Erste Lösung: quadratisch
  - Zweite Lösung: linear

Unifikation

Grundoperation des Berechnungsmodells ist Unifikation
(☞ Typ-Inferenz). Zur Erinnerung

- **Unifikation** = Lösen von Systemen von Term-Gleichungen
  - Lösung = **Substitution** (Unifikator) der Variablen, die die Terme **strukturell gleich** machen.
  - Bsp.:
    - `app([a|[b]],[c|[d]],Ls) = app([X|Xs],Ys,[X|Zs])`
    - Lösung:
      `{X` $\mapsto$ `a, Xs` $\mapsto$ `b, Ys` $\mapsto$ `[c|[d]], Ls` $\mapsto$ `[a|Zs]}`

Unifikation

- Satz: Ein System von Term-Gleichungen hat entweder:
  - **keine** Lösung
  - **eine eindeutige** Lösung
  - **beliebig viele** Lösungen. In diesem Fall gibt es eine allgemeinste Lösung (**mgu**).

Algorithmus zur Berechnung des mgu

θ := $\emptyset$; push(stack, $T_1 = T_2$); failure = false;
while not empty(stack) and not failure do
  hole X = Y vom stack runter
  case
    X ist eine Variable, die in Y nicht auftritt:
      ersetze X durch Y im stack und in θ
      füge $X \mapsto Y$ zu θ hinzu
    Y ist eine Variable, die in X nicht auftritt:
      ersetze Y durch X im stack und in θ
      füge $Y \mapsto X$ zu θ hinzu
    X und Y sind identische Konstanten oder Variablen: continue
    X ist $f(X_1, \ldots, X_n)$ und Y ist $f(Y_1, \ldots, Y_n)$ mit f=Funktor:
      push(stack, $X_1 = Y_1, X_2 = Y_2, \ldots, X_n = Y_n$)
    sonst:
      failure := true
if failure then output *failure* else output θ

Der *occurs check* Test

```
θ := ∅; push(stack, T1 = T2); failure = false;
while not empty(stack) and not failure do
   hole X = Y vom stack runter
   case
      ....
      X ist eine Variable, die in Y nicht auftritt:
         ersetze Y durch X im stack und in θ
         füge X = Y zu θ hinzu
      ....

if failure then output failure else output θ
```

- ...stellt sicher, dass die Unifikation terminiert; Z.B. gibt es keine endliche gemeinsame Instanz von `X` und `[1,X]`;
- wird aus Effizienzgründen in Implementierungen wie Prolog weggelassen.