Erreichbarkeitsrelationen

Aus den DCG-Regeln der Grammatik wollen wir

(i) die *Kettenrelation* $\{ (A, B) \mid A \Rightarrow^+ B \}$,

(ii) die *transitive Hülle der Linke-Ecken-Relation*

$$<_{lc}{}^+ \quad = \quad \{(B, A) \mid \exists\gamma \in Cat^* \ A \Rightarrow^+ B\gamma\}$$

berechnen, jeweils als die transitive Hülle $Basis^+$ einer Relation

$$Basis \subseteq Cat \times Cat \subseteq Term \times Term.$$

Möglichkeit 1 *Cat* erlaubt nur Grundterme von Prolog.

Bedingte DCG-Regeln `(A --> B) :- C` müssen durch Auswerten von `C` in DCG-Regeln `(A' --> B')` ohne Variable übersetzt werden.

Nachteil: *Cat* ist zu groß (für Deutsch-Grammatik), Berechnung von $Basis^+$ stößt an Zeit- und Platzgrenzen![1]

Vorteil: $Basis^+$ besteht aus *Grundfakten*. Schneller Zugriff durch SWI-Prologs `index/1`-Speicherung.

[1] Mit den Merkmalen `a([Kompl,Funkt],[Komp,ADekl,Gen,Num,Kas])` hat man allein $(|\mathit{Präp.}| \times 3) \times (3 \times 3 \times 3 \times 2 \times 4) \simeq 10 \times 3^4 \times 2^3 = 650$ Unterkategorien von Adjektiven.

Möglichkeit 2 *Cat* erlaubt Terme mit freien Variablen.

Vorteil: Eine kleine Relation S zwischen offenen Termen kann als Darstellung einer großen Relation $I(S)$ dienen:

$$I(S) := \{ \langle s,t\rangle \mid \langle s,t\rangle \textit{ ist Instanz eines } \langle u,v\rangle \in S \}$$

Für die Anwendung im Parser genügt es, (mäßige) Vergrößerungen $I(S) \supseteq \mathit{Basis}^+$ durch kleine S darzustellen.

Nachteil: Schwierig zu implementieren:

(i) berechne aus den Grammatikregeln eine Relation S_0 mit $I(S_0) \supseteq \mathit{Basis}$,

(ii) vergrößere $S_0 \subset S_1 \subset \cdots \subset S_n =: S$ möglichst wenig, bis $\mathit{Basis}^+ \subseteq I(S)$ gilt,

(iii) benutze im Parser die als `tabelle/2` gespeicherten (schematischen) Fakten zu S, um $\langle s,t\rangle \in I(S)$ durch `?- tabelle(s,t).` zu testen.

Für (ii): zu $\langle r,s\rangle \in S_i$ und $\langle t,u\rangle \in S_i$ muß man die „allgemeinsten" Instanzen $\langle r',s'\rangle \in I(\langle r,s\rangle)$ und $\langle t',u'\rangle \in I(\langle t,u\rangle)$ mit $s' \equiv t'$ nehmen und dann $\langle r',u'\rangle$ in S_{i+1} aufnehmen, falls es nicht schon in $I(S_i)$ ist, z.B.

$$\frac{\langle f(x,y),y\rangle \in S_i \qquad \langle g(x,y),h(x)\rangle \in S_i}{\langle f(x,g(x',y')),h(x')\rangle \in S_{i+1}} \quad (tc')$$

Das macht man so lange, bis $S_{i+1} = S_i$ ist.

Vorberechnung der Kettenbeziehung

`chain/2` $:= \{(A,B) \mid A \Rightarrow^+ B\}$

Die Paare `(A,A)` werden später separat behandelt.

Voraussetzung: `del/1` (für `deletable/1`) ist geladen.

$\langle$*Kettenbeziehung*$\rangle \equiv$

```
:- dynamic chain/2.

compute_chain_relation :-
        retractall(chain(_,_)),
        ensure_loaded('Parser/transclos.pl'),
        compute_chain_base,
        compute_chain_itc.
```

Basis: $\{\, \langle A, B\rangle \in Cat^2 \mid (A \to \alpha B \gamma) \in P,\ \alpha\gamma \Rightarrow^* \epsilon \,\}$

$\langle$*Kettenbeziehung, Basis*$\rangle \equiv$

```
compute_chain_base :-
    clause(A --> Body,Cond), not(A = ?(_)),
    tupleToList(Body,BodyL),
    append(Alpha,[B|Gamma],BodyL),
    ((var(B) ; B = {Goal}) -> true
    ; deletables(Alpha),
        deletables(Gamma),
        addChains(A,B,Conds)
    ),
    fail.
compute_chain_base.

deletables([C|Cs]) :-
        deletable(C), deletables(Cs).
deletables([]).
```

`addChains(A,B,Cond)` untersucht `B` nach seiner Struktur und übernimmt die erreichbaren Teile in die Kettenbeziehung:

Ist `B` eine Variable, so ist `B` ein Objekt unbekannter Kategorie und hat einen nicht löschbaren Kopf-Nachbarn:

⟨*Kettenbeziehung*⟩+≡
```
addChains(A,B,Cond) :- var(B),!.
```

`deletable(D)` belegt i.a. Variable und `simplify/2` wertet variablenfreie Teilbedingungen von `Cond` aus, sodaß meist Fakten zu `chain/2` hinzukommen.

⟨*Kettenbeziehung*⟩+≡
```
addChains(A,(C,D),Cond) :-
        !, ( deletable(D), simplify(Cond,NewCond),
               addChains(A,C,Cond)
           ; deletable(C), simplify(Cond,NewCond),
               addChains(A,D,Cond)
           ).
addChains(A,(C;D),Cond) :-
        !, addChains(A,C,Cond), addChains(A,D,Cond).
addChains(A,?(B),Cond) :-
        !, addChains(A,B,Cond).
addChains(A,[W|Ws],Cond) :-
        !, add(chain(A,[W|Ws]) :- Cond).
addChains(_A,[],_Cond) :-       !.
addChains(_A,{_Goal},_Cond) :- !. % tritt nicht auf.
```

Kettenregeln `(A --> B) :- Cond.` werden *bedingte* Tupel von `chain/2` (kompakter als Fakten):

⟨*Kettenbeziehung*⟩+≡
```
addChains(A,B,Cond) :-
```

```
          (A == B -> true
          ; simplify(Cond,NewCond),
              add(chain(A,B) :- NewCond)).
```

Die Basisrelation enthält nur Paare ⟨`A`, `B`⟩, bei denen `A` ≠ `B` nicht von der Form `(C;D)`, `(C,D)`, `[C|Cs]`, `[]`, `?(C)`, `{C}` sind.

Aufg. 4 Warum kann man auf Kategorien `?A` bei der folgenden Bildung der irreflexiven transitiven Hülle verzichten?

Abschluß: Mit `itc/3` wird die irreflexive transitive Hülle der Basisrelation `chain/2` gebildet. Aus `chain/2` werden die Elemente der Form `chain(X,X)` entfernt und dann die neu gefundenen hinzugefügt:

⟨*Kettenbeziehung*⟩+≡

```
  compute_chain_itc :-
      findall((X,Y),itc(chain,X,Y),Chains),
      retractall(chain(X,X)),
      add_chain(Chains).

  add_chain([(X,Y)|T]) :-
      add(chain(X,Y)),
      add_chain(T).
  add_chain([]).
```

Klauseln der Grundrelation werden nicht entfernt, um eine kompaktere Darstellung zu erreichen.

Bem. `chain/2` enthält keine Paare mit Komponenten `A = B`: das wird durch `chain(A,B)`'s im Parser berücksichtigt.

`chain/2` hat keine Paare mit Komponenenten `?B` oder `(B;C)`: das wird durch `instance(B,A)`'s im Parser berücksichtigt.

Verwendung von $B \Rightarrow^* D$ im Parser

Um an eine Erwartungskategorie `B` einen vollständigen Baum mit Wurzel `B` durch eine (kanonische) Kettenableitung anzuschließen, bzw. um diese Anschließbarkeit festzustellen, dienen:

⟨*connect*⟩≡

```
connect(B,D,DTree,BTree) :-
        instance(BI,B),
        ( BI = D, BTree = DTree
        ; chain(BI,D), BTree = [BI,DTree]
        ).
connect(B,D) :-
        instance(BI,B),
        ( BI = D ; chain(BI,D) ).
```

Die atomaren Instanzkategorien `D` einer zusammengesetzten Kategorie `C` werden durch `instance(D,C)` bestimmt:

⟨*instance*⟩≡

```
instance(D,X) :- var(X), !, fail.
instance(D,(A,B)) :-  % für A oder B ein {Goal}
        !,( deletable(A), instance(D,B)
          ; deletable(B), instance(D,A)
          ).
instance(D,(A;B)) :-
        !, (instance(D,A); instance(D,B)).
instance(D,?(A)) :-
        !, instance(D,A).
instance(D,{_Goal}) :-
        !, fail.
instance(D,D).
```

Bem.:

(i) Der Fall der Konjunktion ist nur für `(A,{Cond})` und `({Cond},B)` gedacht; echte Konjunktionen von Konstituenten werden nicht behandelt. Die Parser sind also nicht für EBNF-Grammatikregeln mit expliziter Verwendung von Konjunktion wie in `A --> ((B,C),D)` geeignet; Rechtsklammerung `(B,(C,D))` wird in Listen `[B,C,D]` aufgelöst und ist daher kein Problem.

(ii) Da `(B;C)` und `?(B)` nicht als Kategorien erkannter Teilphrasen auftreten, kann man Tupel $\langle \mathtt{A}, ?\mathtt{B} \rangle$ und $\langle \mathtt{A}, (\mathtt{B}; \mathtt{C}) \rangle$ aus der Basis der Kettenrelation herauslassen.

(iii) Da `(A;B)` nicht als Kopf einer Regel auftritt, kommen keine Tupel $\langle (\mathtt{A}; \mathtt{B}), \mathtt{C} \rangle$ in der Relation vor. Damit man `?A` als Kopf von Löschregeln erlauben kann, nehmen wir die Tupel $\langle ?\mathtt{A}, \mathtt{B} \rangle$ nicht in die Grundrelation auf; sie werden später durch `instance(A,?A)` erfaßt.

Berechnung der Linke-Ecke-Relation
$\mathit{first} := \{(B, A) \mid \exists\beta\ A \Rightarrow^+ B\beta\}$

⟨*Vorberechnung der First-Relation*⟩≡

```
:- dynamic first/2.

compute_first_relation :-
        retractall(first(_,_)), % alte Tabelle
        compute_first_base,
        compute_first_itc.
```

Nachdem ggf. eine alte Tabelle gelöscht ist, wird aus den Grammatikregeln die Grundrelation

$$\mathit{Basis} := \{\, (B, A) \in \mathit{Cat}^2 \mid (A \to \alpha B\beta) \in P,\ \alpha \Rightarrow^* \epsilon \,\}$$

berechnet und durch (schematische) Fakten dargestellt:[2]

⟨*Vorberechnung der First-Relation, Basis*⟩≡

```
compute_first_base :-
        (A --> Gamma),
        collect_first_base(A,Gamma),
        fail.
compute_first_base.
```

In `(A --> Gamma)` werden löschbare Präfixe `Alpha` von `Gamma` = `Alpha B Beta` gesucht; dabei wertet `deletables(Alpha)` die `{Goal}` in `Alpha` aus und belegt Variable. `B` wird rekursiv zerlegt und seine atomaren(!) Teile `C` als `first(C,A)` gespeichert.

Tupel `(C,A)` mit zusammengesetztem `C` werden mit `instance/2` im Parser auf die mit atomarem `C` reduziert.

[2]Darstellung durch Regeln: als Option `first_base(rules)`.

`collect_first_base`

Beim Sammeln der Tupel durch `collect_first_base` werden die rechten DCG-Regelseiten zerlegt, bis man auf eine atomare Kategorie stößt.

⟨*Vorberechnung der First-Relation, Basis*⟩+≡

```
collect_first_base(_A,B) :-
        var(B),!,
        write(user,'\nFEHLER: variable Kategorie\n').
collect_first_base(?A,_) :- !.
collect_first_base((A;B),_) :- !.

collect_first_base(A,(B,C)) :-
        !,collect_first_base(A,B),
        deletable(B),     % belegt Variable!
        collect_first_base(A,C).
collect_first_base(A,(B;C)) :-
        !, collect_first_base(A,B),
        collect_first_base(A,C).
collect_first_base(A,?(B)) :-
        !, collect_first_base(A,B).
collect_first_base(_A,[]) :-
        !.
collect_first_base(_A,{_Goal}) :-
        !.
```

Bem.: Bei `collect_first_base(A,(B,C))` muß man *jede* Belegung betrachten, die `deletable(B)` beweisbar macht. Die aus `(A,B)` stammenden Tupel sind schon gespeichert und bleiben unverändert, wenn die *darin* freien Variablen durch den Test `deletable(B)` instantiiert werden.

`collect_first_base`, **Forts.**

Da i.a. die Relation $Basis^+$ (oder ihre Berechnung) sehr aufwendig ist, darf man beim Speichern die Tupel vereinfachen:

Option `first_base(explicit)`: Die Basis soll aus den *explizit* an den `(A --> Alpha)` abgelesenen Tupeln bestehen.

Option `first_base(restrict)`: Die Basis soll stattdessen die expliziten Tupel durch Weglassen von Teiltermen vereinfachen, d.h. Kategorienausdrücke auf gewisse Merkmale *einschränken.* Dazu ist ein Prädikat `restrict(Cat+,CatR-)` durch Lexikon und Grammatik zu definieren.

$\langle$ *Vorberechnung der First-Relation, Basis* $\rangle{+}{\equiv}$

```
collect_first_base(A,[B|Rest]) :-
        !,(first_base(restrict)
          -> restrict_all([A,B|Rest],[AR,BR|RestR]),
              add(first([BR|RestR],AR))
          ; add(first([B|Rest],A))).
collect_first_base(A,B) :-
        !,(A == B -> true % ignoriere (A,A)
          ; (first_base(restrict)
            -> restrict_all([A,B],[AR,BR]),
                add(first(BR,AR))
            ; add(first(B,A)))).

restrict_all([T|Ts],[TR|TsR]) :-
        restrict(T,TR),
        restrict_all(Ts,TsR).
restrict_all([],[]).
```

`collect_first_itc`

Vor.: Die Grammatik erlaubt nur Kategorienausdrücke einer bestimmten Größe. Regeln wie `cat(f(X)) --> cat(X)` mit unbeschränktem `X` werden dadurch ausgeschlossen. Sonst kann die Vorberechnung divergieren.

Abschluß: Mit `itc/3` wird die irreflexive transitive Hülle der Basisrelation `first/2` gebildet. Zur Basisrelation werden die neu gefundenen Tupel `(X,Y)` als `first(X,Y)` hinzugefügt:

⟨*Vorberechnung der First-Relation, Abschluß*⟩≡

```
compute_first_itc :-
    findall((X,Y),itc(first,X,Y),Firsts),
    add_first(Firsts).

add_first([(X,Y)|Rest]) :-
    add(first(X,Y)), add_first(Rest).
add_first([]).
```

Die irreflexive transitive Hülle $Basis^+$ speichern wir als Tabelle `first/2` in der Prolog-Datenbank. Elemente der Form `(X,X)` werden im Parser durch `instance/2` mitbehandelt.

Bem.: Diese Relation wird in der Literatur auch `leftcorner/2` genannt; dann ist `first/2` der Teil, bei dem die erste Komponente eine lexikalische Kategorie ist.

Berechnung der transitiven Hülle einer Relation

Die transitive Hülle einer Relation wird mit dem Algorithmus von Warshall berechnet. Die Version aus dem Prolog-Buch von R.O'Keefe geht *nur für eine Basisrelation aus variablenfreien Termen.*

Für Grammatikregeln, deren freie Variable nicht durch Bereichsprädikate beschränkt sind, habe ich zwei weitere Möglichkeiten implementiert (vgl. Doku/Anhang Hilfsprogramme):

(i) eine Variante des Warshall-Algorithmus, die Terme mit Variablen so behandelt, als seien diese ∀-quantifiziert,

(ii) das Programm `itc/4`, das die transitive Hülle so berechnet, wie unter **Möglichkeit 2** skizziert wurde.

⟨*Parser/transclos.pl*⟩≡

```
itc(Base,Ancestor,Descendant) :-
        tc(Base,Ancestor,Descendant),
        not(Ancestor == Descendant).
```

`tc(Base,X,Y)` ist die transitive Hülle der Relation `Base/2`.

Diese Programme können bei größeren Grammatiken immer noch an Speichergrenzen scheitern. Man *muß* dann

(i) die Basisrelation durch die Auswahl `first_base(restrict)` vereinfachen, und

(ii) mit `predictor(uniform)` den uniformen statischen Predictor benutzen (s.u.).

Die Details werden hier nicht dargestellt (vgl. Dokumentation).

Vorberechnungen:

Beisp. 12 Betrachte folgende DCG mit einer schematischen Regel ('Metaregel'):

⟨*dcg.3.grm*⟩≡
```
:- op(1100,xfx,'-->').

start_symbol(s(X)).

(s(X) --> a(X), (b(X) ; d(X), {X=3})) :-
    member(X,[1,2,3]).
b(2) --> [].
d(X) --> b(X), c(X).

word(a,a(1)).     word(b,b(1)).     word(c,c(1)).
word(a,a(2)).     word(b,b(2)).     word(c,c(2)).
word(a,a(3)).     word(b,b(3)).     word(c,c(3)).
```

Die Übersetzung der Grammatik berechnet folgende Tabellen:

⟨*Vorberechnete Tabellen*⟩≡
```
del(b(2)).

chain(s(2), a(2)).     chain(d(2), c(2)).

first(a(1), s(1)).     first(b(A), d(A)).
first(a(2), s(2)).     first(c(2), d(2)).
first(a(3), s(3)).
```

Die `first`-Tabelle enthält Terme mit freien Variablen, da die Grammatik eine Regel mit einer freien Variable enthält (die nicht durch eine Bereichsbedingung beschränkt ist).