Prolog Implementierung (Interpreter)

SLD-Resolution (Wiederholung):

- Aktuelle Anfrage: $A_1 \wedge A_2 \wedge \cdots \wedge A_n$.
  Falls $n = 0$: Leere Konjunktion entspricht „true“, Anfrage bewiesen.
- Wähle Regel: $B_0 \leftarrow B_1 \wedge \cdots \wedge B_m$.
- Nenne Variablen um: $B'_0 \leftarrow B'_1 \wedge \cdots \wedge B'_m$.
- Berechne allg. Unifikator θ von A_1 und B'_0.
  Abbruch bzw. Backtracking falls nicht unifizierbar: Irgendwann vorher wurde eine falsche Regel gewählt oder die Anfrage ist nicht beweisbar.
- Neue Anfrage: $B'_1\theta \wedge \cdots \wedge B'_m\theta \wedge A_2\theta \wedge \cdots \wedge A_n\theta$.

Probleme:

- Alte Anfragen werden für Backtracking noch gebraucht. Vollständiges Kopieren zu ineffizient.
  Wir ignorieren dieses Problem zunächst und nehmen an, daß wir immer die richtige Regel raten. Backtracking erst in der 3. Ausbaustufe.
- Es ist zu aufwendig, die Substitution θ in jedem Schritt auf die ganze Anfrage anzuwenden.
  Man sollte $A_2 \wedge \cdots \wedge A_n$ überhaupt nicht anfassen müssen. Zunächst nur Aussagenlogik, Argumente/Substitutionen erst in 2. Ausbaustufe.
- Kopieren von Programmcode (Regel) vermeiden.
  Dies wäre beim Umbennen der Variablen nötig (2. Ausbaustufe), aber auch schon bei der Konstruktion der neuen Anfrage (jetzt).

Aussagenlogik ohne Backtracking (1)

Beispiel/Aufgabe:

- Geben Sie eine SLD-Ableitung für p an:

$$p \leftarrow q \wedge r.$$
$$q \leftarrow s \wedge t.$$
$$r.$$
$$s.$$
$$t.$$

Datenstrukturen:

- Literal: Prädikat (später plus Argumente).
  type literal = **record** pred: string; **end**;
  Statt Prädikatnamen in jedem Literal besser Index in Prädikattabelle.
- Anfrage: Liste von Literalen.
  type goal = **record** first: literal; next: ↑goal **end**;
- Regel: Paar aus Regelkopf und Regelrumpf.
  type rule = **record** head: literal; body: ↑goal **end**;

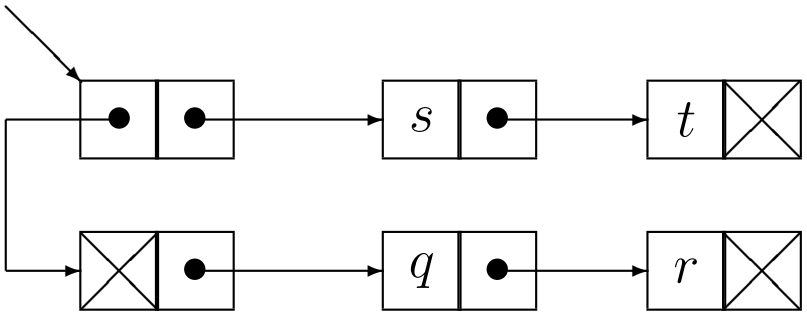
Beispiel/Aufgabe (Forts.):

- Zeichnen Sie die Datenstrukturen für die Anfrage $q \wedge r$ and die Regel $q \leftarrow s \wedge t$. Wie wird die neue Anfrage konstruiert? Was muß kopiert werden?

Aussagenlogik ohne Backtracking (2)

Idee:

- Repräsentiere die aktuelle Anfrage als Listen von Listen, z.B. $s \wedge t \wedge r$:



- Beachte: Bei allen Teillisten außer der ersten muß das erste Literal noch gelöscht werden.

  Beim Resolutionsschritt wird ja das erste Literal der Anfrage gelöscht und vorne eine neue Teilliste angefügt. Wir führen die Löschung aber erst später aus, wenn wir auf eine hintere Teillisten wirklich zugreifen. Das ist später für das Backtracking nützlich. Es paßt aber in das Prinzip, soviel wie möglich Arbeit in die Zukunft zu verschieben (die wegen einem vorherigen Abbruch vielleicht nicht kommt). Die „Listen von Listen“ Methode verschiebt auch nur die Konkatenations-Arbeit.

Beispiel/Aufgabe (Forts.):

- Skizzieren Sie die Ausführung (Aufrufstack, PC) eines strukturell ähnlichen Programms in einer imperativen Programmiersprache (z.B. Pascal).

Aussagenlogik ohne Backtracking (3)

Datenstruktur:

- Stack von Teillisten.

  Entspricht Stack von „Activation Records“ in imperativen Sprachen.
- **type** act = **record** parent: ↑act; query: ↑goal **end**;
  push(top); **begin** a := **new**(act); a↑.parent := top; **return**(a); **end**;
  pop(top); **begin** a := top↑.parent; **dispose**(top); **return**(a); **end**;

Algorithmus:

- Initialisierung, Hauptschleife:

  stacktop := push(**nil**);
  stacktop↑.query := user_query; % Gegebene Anfrage
  while true do % Wiederhole, bis expliziter Abbruch
- Suche Arbeit (nächstes zu beweisendes Literal):

  while stacktop↑.query = **nil do**
  stacktop := pop(stacktop);
  if stacktop = **nil then print**(“yes”); **exit**();
  stacktop↑.query := stacktop↑.query↑.next;
- Wähle passende Regel:

  rule := select_rule(stacktop↑.query.first);
  if not unify(stacktop↑.query.first, rule.head) **then abort**();
  % „unify“ testet nur die Gleichheit der 0-stelligen Prädikate.
- Lege Regelrumpf auf Stack:

  stacktop := push(stacktop);
  stacktop↑.query := rule.body;

Term-Repräsentation (1)

Grundidee:

- Regeln können mehrfach aktiviert werden (durch Rekursion). Sie sollen aber nicht kopiert werden.
- Lösung: Repräsentiere Variablen durch Offsets. Für jede Regelaktivierung wird ein Array mit den Variablen-Werten angelegt (Variablen-Frame).
- Ein Term ist dann bestimmt durch
  - einen Zeiger auf den „Prototyp“ in einer Regel
  - und einen Zeiger auf einen Variablen-Frame.

„Structure Sharing“ Methode:

- Da die Werte im Variablen-Frame wieder Terme sind, ist es naheliegend, auch dort Zeiger-Paare auf Prototyp und Variablen-Frame einzutragen.
- **type** prototype = **record case** tag **of**
  var: (offset: integer);
  fun: (functor: sym); % plus Argumente, s.u.

  end;

  type term = **record** proto: prototype; frame: ↑varframe **end**;

  type varframe = **array of** term;
- Die Argumente von Funktoren (bzw. Zeiger darauf) werden meist im Speicher (Array) für Prototypen direkt hinter dem Funktor abgelegt.

Term-Repräsentation (2)

Beachte:

- Der Wert einer Variablen kann wieder eine gebundene Variable sein u.s.w. (Pointerketten).

  Z.B.: ?- p(X). p(Y) :- q(Y). q(Z) :- r(Z). r(a).
- Dies entspricht dem Verschieben der Komposition von Substitutionen auf den Zeitpunkt des Zugriffs auf das Ergebnis.

Beispiel/Aufgabe:

Erläutern Sie die Unifikation (mit Zeichnung):

- append(.(#1,#2), #3, .(#1,#4)) :-
  append(#2, #3, #4).
  append([], #1, #1).
- ?- append(.(a, .(b, [])), .(c, []), #1).

Problem:

- Variablen-Frames können nicht wie bei einer imperativen Sprache freigegeben werden.

  D.h. am Ende eines Prädikataufrufes. Der Grund ist, daß tief in der Rekursion konstruierte Terme „hochgereicht“ werden können.
- Also nur beim Backtracking freigeben. Besser: Regeln analysieren, Variablen-Frames in lokalen und globalen Teil spalten: Variablen, die nicht Wert anderer Variablen werden können, am Aufrufende freigeben.

Term-Repräsentation (3)

„Copy On Use“ Methode:

- Hier gibt es zwei Arten von Termen:
  Es ist ein hybrides System, nicht das Gegenteil von Structure Sharing.
- Virtuelle Terme sind durch Prototyp mit Offsets und Variablen-Frame repräsentiert.
  Wie oben, aber im Variablen-Frame stehen Pointer auf reale Terme, keine (Prototyp,Frame)-Paare.
- Reale Terme enthalten für die Variablen Speicherzellen mit einem Pointer auf den Wert.
- Wenn ein virtueller Term Wert einer Variablen werden soll, wird eine reale Kopie erstellt.
- Reale Terme werden auf dem Heap abgespeichert, Variablen-Frames auf dem Aufruf-Stack.
  Dies entspricht gewissermaßen einer dynamischen Unterteilung in lokale und globale Variablen (s.o.).
- Wenn man also dafür sorgt, daß Variablen-Variablen-Bindungen immer nach unten im Stack gehen, und der Heap keine Referenzen in den Stack enthält, dann kann man die Variablen-Frames am Ende des Prädikat-Aufrufs freigeben (sofern kein Backtracking mehr möglich ist).
  Beispiel: `m :- p(X).` `p(Y) :- q(Y,Z).` `q(W,W).`
- Moderne Prolog-Systeme verwenden meist „Copy on Use“.
- Beispiel/Aufgabe: Wie würde „append“-Beispiel mit „Copy on Use“ ausgeführt? Zeichnen Sie die Datenstrukturen!

Term-Repräsentation (4)

Unifikation:

- **type** termInfo = **record case** tag **of**
  var: (pointer: term);
  off: (offset: integer); % Nur in virtuellen Termen
  fun: (functor: sym);
  end;
  type term = ↑termInfo; % Oder Index in großes Array
- **function** deref(term, frame): term
  if isoff(term) **then** term := frame + term↑.offset;
  while isvar(term) **and** term↑.pointer <> „unbound“ **do**
  term := term↑.pointer;
  return(term);
- **procedure** bind(var, term, frame)
  var↑.pointer := copy(term, frame);
  % Falls der Term schon ein realer Term ist: nicht kopieren.
  % Beachte Richtung von Variablen-Variablen Bindungen!
- **function** unify(term1, frame1, term2, frame2): bool
  t1 := deref(term1, frame1); t2 := deref(term2, frame2);
  if t1=t2 **then return**(**true**);
  % Damit bei Unifikation X=X Variable ungebunden bleibt.
  % Oder: ungebundene Variablen = Pointer auf sich selbst.
  else if isvar(t1) **then** bind(t1, t2, frame2); **return**(**true**);
  else if isvar(t1) **then** bind(t2, t1, frame1); **return**(**true**);
  else if t1↑.functor <> t2↑.functor **then return**(**false**);
  else n := t1↑.functor↑.arity; % oder t2, sind ja gleich.
  for i := 1 **to** n **do**
  if not unify(t1+i, t2+i) **then return**(**false**);
  return(**true**);

Backtracking (1)

Allgemeines:

- Es kann mehrere anwendbare Regeln geben.
- Dann: Zustand sichern, erste Regel ausprobieren.

  Der Systemzustand besteht hauptsächlich aus dem Stack mit den verketteten Teilanfragen (Activation Records) und den Variablen-Frames.
- Falls später Unifikation scheitert: Systemzustand wiederherstellen, nächste Regel ausprobieren.
- Allgemein entsteht ein Stack gesicherter Zustände, weil beim Ausprobieren einer Regel weitere Alternativen auftreten können.

Grundidee:

- Sichern/Wiederherstellen muß schnell gehen.

  Man kann also nicht jedesmal eine vollständige Kopie anlegen.
- Idee: Friere aktuellen Stackinhalt ein — man darf zwar oben noch etwas drauflegen, aber nichts herunternehmen oder ändern.
- Ausnahme: Variablen dürfen gebunden werden, aber ihre Adresse muß dann auf einem speziellen Stack („Trail“) gemerkt werden. Ebenso für Heap.

  Die Adresse reicht aus, der alte Wert war ja „unbound“. Beim Sichern merkt man sich den Trailpointer, so daß man beim Wiederherstellen bis zu diesem Punkt die Variablen wieder ungebunden machen kann.

Backtracking (2)

Rückkehr zu altem Activation Record:

- Falls in nicht eingefrorenen Teil: wie bisher.

  D.h. Platz darüber freigeben, „query“-Feld auf nächstes Literal setzen.

- Sonst: Kopie des Activation Records oben auf dem Stack anlegen.

  Natürlich möchte man den zugehörigen Variablen-Frame nicht mit kopieren. Deswegen wird auf den Variablen-Frame in Prolog über einen Zeiger im Activation Record zugegriffen, nicht über eine feste Distanz. Tatsächlich kann man aber den „parent“-Zeiger dafür benutzen.

Implementierung von Zustandssicherungen:

- Der Zustand wird gesichert, indem ein sogenannter „Choice-Point“ auf dem Stack abgelegt wird. Alles darunter gilt dann als eingefroren.
- Der Choicepoint enthält die Werte der globalen Variablen: nächste anwendbare Regel, Stackpointer, Trailpointer, Heappointer (beim Backtracken werden die angelegten Termkopien gelöscht), letzter Choicepoint, aktueller Aktivation-Record.

Beispiel/Aufgabe:

Erläutern Sie die Ausführung dieses Programmes bei der Anfrage $p(X)$:
$p(X) : -q(X), r(X).$
$q(X) : -s(X).$
$s(a).$
$s(b).$
$r(b).$