Sortieren (1)

Quicksort:

- „Divide and Conquer“: Liste aufteilen in kleinere/größere Elemente als ein Trennelement.
- ```
  qsort([], []).
  qsort([T|L], S) :-
          partition(L, T, L_kleiner, L_größer),
          qsort(L_kleiner, S_kleiner),
          qsort(L_größer, S_größer),
          append(S_kleiner, [T|S_größer], S).
  ```
- ```
  partition([], _, [], []).
  partition([E|L], T, [E|L_kleiner], L_größer) :-
          E < T, partition(L, T, L_kleiner, L_größer).
  partition([E|L], T, L_kleiner, [E|L_größer]) :-
          E >= T, partition(L, T, L_kleiner, L_größer).
  ```

Verbesserung (ohne explizites append):

- ```
  qsort(L, S) :- qsort(L, [], S).    % qsort(L, X, S) <=> S = sort(L) o X
  qsort([], S, S).
  qsort([T|L], S_noch_größer, S) :-
          partition(L, T, L_kleiner, L_größer),
          qsort(L_größer, S_noch_größer, S_größer_und_noch_größer),
          qsort(L_kleiner, [T|S_größer_und_noch_größer], S).
  ```
  ```

Sortieren (2)

Mergesort:

- Liste in einelementige Listen aufteilen, dann jeweils zwei sortierte Listen zu einer „mischen“.
- ```
  mergesort([], []).
  mergesort(L, S) :- lists(L, LL), msort(LL, [S]).
  ```
- ```
  lists([], []).
  lists([E|L], [[E]|LL]) :- lists(L, LL).
  ```
  Verbesserung: Sortierte Teilstücke nicht aufspalten.
- ```
  msort([], []).
  msort([S], [S]).
  msort([S1,S2|SL], S) :-
          merge(S1, S2, M),
          msort(SL, ML),      % Restliche Paare mischen
          msort([M|ML], S).   % Nächster Durchlauf
  ```
- ```
  merge([], L, L).
  merge(L, [], L).
  merge([E1|R1], [E2|R2], [E1|M]) :-
          E1 < E2, merge(R1, [E2|R2], M).
  merge([E1|R1], [E2|R2], [E2|M]) :-
          E1 >= E2, merge([E1|R1], R2, M).
  ```

Suchen in Symboltabellen

Offene Liste:

- Es ist eine Abbildung von Schlüsseln auf Werte aufzubauen.
- Repräsentation: Liste von Schlüssel-Wert Paaren mit einer ungebundenen Variable am Ende.

  Zum Beispiel: [keyval(stefan,4953), keyval(michael,4960)|X].
- Suchen eines Eintrags und ggf. einfügen:
  ```
  lookup([keyval(K,V1)|_], K, V2) :- !, V1 = V2.
  lookup([_|R], K, V) :- lookup(R, K, V).
  ```

Alternativen:

- Suchbaum, Variablen als Blätter:
  ```
  lookup(node(K1,V1,L,R), K2, V2) :-
        K1 = K2, !, V1 = V2.
  lookup(node(K1,_,_,R), K2, V) :-
        K1 < K2, !, lookup(R, K2, V).
  lookup(node(K1,_,L,_), K2, V) :-
                      lookup(L, K2, V).
  ```
- Faktensammlung, z.B. keyval(stefan, 4953).

  Aber `assert` ist „nichtlogisch“ und sehr aufwendig.

Differenz-Listen

Allgemeines:

- Man repräsentiere die Liste [1,2] als [1,2|X]-X.

  X kann dabei natürlich an einen Wert gebunden werden, Spezialfälle sind also [1,2]-[] und [1,2,3,4]-[3,4].

- Konkatenation ist in konstanter Zeit möglich.

  Falls der Rest der ersten Liste ungebunden ist.

- Wenn man in Pascal Listen in konstanter Zeit aneinanderhängen will, braucht man ja auch einen Pointer auf das Ende der Liste.

Beispiel-Anwendungen:

- append(X-Y, Y-Z, X-Z).
- reverse(X,Y) :- rev(X, Y-[]).

  rev([], Y-Y).

  rev([E|R], Y-Z) :- rev(R, Y-[E|Z]).

Probleme:

- append([1,2]-[2], [3]-[], X). ⟶ „no" statt [1,3].

  „append" funktioniert nur für „kompatible" Differenzlisten, d.h. für Listen A-B und C-D, bei denen B und C unifizierbar sind.

- empty(X-X).   ?- empty([1|Y]-Y). ⟶ „yes".

  Hier fehlt der Occur-Check.

Suche in einem Zustandsraum

Fährmann-Rätsel:

- Es sollen ein Wolf, eine Ziege und ein Kohlkopf übergesetzt werden, aber das Boot faßt außer dem Fährmann nur einen „Passagier". Wolf und Ziege sowie Ziege und Kohlkopf dürfen nicht allein an einem Ufer zurückbleiben.
- Die Zustände werden als Quadrupel mit den Positionen (`l`, `r`) von Wolf, Ziege, Kohlkopf und Fährmann repräsentiert.

  zug(zust(l,Z,K,l), zust(r,Z,K,r), hin(wolf)).
  zug(zust(r,Z,K,r), zust(l,Z,K,l), zurück(wolf)).

  Entsprechend noch für Ziege und Kohlkopf.

- illegal(zust(l,l,_,r)). illegal(zust(r,r,_,l)).
  illegal(zust(_,l,l,r)). illegal(zust(_,r,r,l)).
- Es ist wichtig, über die schon besuchten Zustände Buch zu führen (sonst Endlosschleife).

  ```
  erreichbar(Zust, _, [], Zust) :- !.
  erreichbar(Zust1, Alt, [Zug|Züge], Zust3) :-
      zug(Zust1, Zust2, Zug),
      not(illegal(Zust2)),
      not(member(Zust2, Alt)),
      erreichbar(Zust2, [Zust2|Alt], Züge, Zust3).
  ```

- ?- erreichbar(zust(l,l,l,l), [], Lösung, zust(r,r,r,r)).

Meta-Programmierung

Ein Meta-Interpreter für Prolog:

- Metaprogramme behandeln andere Programme als Daten.
- In Prolog besonders einfach, da Programme und Daten beides Terme sind.
- Repräsentation für Klauseln:

  rule(p(a), true). p(a).

  rule(q(X,Y), (p(X), p(Y))). q(X,Y) :- p(X), p(Y).
- Meta-Interpreter:

  solve(true).

  solve((Lit, Conj)) :- solve(Lit), solve(Conj).

  solve(Head) :- rule(Head, Body), solve(Body).
- Es fehlen noch die eingebauten Prädikate.

Ausbaumöglichkeiten/Anwendungen:

- Beweisbaum (Erklärung) berechnen.
- Mit Wahrscheinlichkeiten rechnen.

  Dies ist bei unsicherem Wissen (Faustregeln) interessant.
- Vom Benutzer erfragbare Fakten einführen.