Logisch programmeren 2011

Week 3: werken met incomplete datastructuren

1. Incomplete datastructuren

Een krachtige programmeertechniek is het gebruik van incomplete datastructuren: datastructuren die variabelen bevatten.

Operaties op incomplete datastructuren kunnen zo optimaal de ingebouwde unificatie uitbuiten.

In deze cursus komen verschillende voorbeelden aan de orde:

- getallen: verschilgetallen in successor notatie
- lijsten: verschillijsten
- bomen: incomplete bomen
- ...

2. Verschilgetallen

Stel dat we een natuurlijk getal n voorstellen als het verschil tussen twee getallen. Zo zou je 2 kunnen voorstellen als $2-0$, $3-1$, $4-2$, $\ldots$

Klein	$=$	Groot	$-$	Verschil	
s(s(0))		s(s(0))		0	
		s(s(s(0)))		s(0)	
		s(s(s(s(0))))		s(s(0))	
		s(s(X))		X	$\leftarrow$ incompleet!

Een incomplete datastructuur die al deze gevallen dekt is s(s(X))-X; de incomplete datastructuur voor zero wordt X-X. Optellen voor deze verschilgetallen kan worden weergegeven met een prolog feit!

```
plusDif(A-B,B-C,A-C).
```

3. Verschilgetallen: illustratie

Het predicaat d2s/3 hieronder zet een getal in decimale notatie om in een verschilgetal in Peano notatie.

```
d2s(0,X,X).
d2s(N,s(X),Y) :- N>0,
   N1 is N-1,
   d2s(N1,X,Y).
```

We schrijven een evaluator voor symbolische somexpressies waarbij optellen door unificatie van verschilgetallen wordt vervangen. Een somexpressie is (i) een natuurlijk getal of (ii) een uitdrukking van de vorm $E_1 + E_2$ waar E_1 en E_2 somexpressies zijn.

```
eval(N,D,D1) :- integer(N), N>=0, d2s(N,D,D1).

eval(N+M,D,D1) :-   % immers: plusDif(D-D0,D0-D1,D-D1)
   eval(N,D,D0),    % evaluatie van N: D-D0
   eval(M,D0,D1).   % evaluatie van M: D0-D1
```

4. Verschillijsten

Stel dat we een lijst representeren als het verschil van twee lijsten. Bijvoorbeeld: de lijst [a,b] wordt [a,b]-[], of [a,b,c]-[c] of ....

De incomplete datastructuur [a,b|Rest]-Rest is unificeerbaar met al deze gevallen.

Met deze incomplete datastructuur kan concatenatie van lijsten aan de ingebouwde unificatie-operatie worden overgelaten:

```
appendDif(A-B,B-C,A-C).
```

Expliciete aanroep van appendDif/3 (zoals plusDif/3 hierboven) kan je wegcompileren als je met verschilstructuren werkt en als de waarden van de eerste twee argumenten bekend zijn op het moment van aanroep.

5. Voorbeeld: quicksort

Het bekende quicksort algoritme werkt als volgt.

- Kies een willekeurig element van de te sorteren lijst.
- Verdeel de rest in elementen kleiner dan het gekozen element, en elementen groter of gelijk aan het gekozen element.
- Sorteer die lijst van kleinere elementen, en de lijst van grotere elementen m.b.v. het quicksort algoritme.
- Concateneer de gesorteerde lijst van kleinere elementen met het gekozen element en de gesorteerde lijst van grotere elementen voor het eindresultaat.

6. quicksort/2: naief

In de Prolog versie hieronder kiezen we de kop van de te sorteren lijst als vergelijkingselement. split(Key,List,Smaller,Bigger) laten we aan de verbeelding over.

```
quicksort( [], []).
quicksort( [X|Tail], Sorted)  :-
   split( X, Tail, Small, Big),
   quicksort( Small, SortedSmall),
   quicksort( Big, SortedBig),
   append( SortedSmall, [X|SortedBig], Sorted).    % ??$#%$#!!
```

Op het moment waarop append/3 wordt aangeroepen zijn SortedSmall en SortedBig bekend. We kunnen de aanroep van append/3 dus vermijden door gebruik te maken van verschillijsten.

7. quicksort/3: verschillijsten

We kunnen een term Lijst-Verschil gebruiken, of Lijst en Verschil als aparte argumenten behandelen.

Hier de definitie van quicksort/2 in termen van quicksort/3 met interpretatie quicksort(Lijst,Gesorteerd,Verschil).

```
quicksort( [], L, L). % L-L is lege lijst
quicksort( [X|Tail], L, L1)  :-
   split( X, Tail, Small, Big),
   quicksort( Small, L, [X|L0]),
   quicksort( Big, L0, L1).
```

De wrapper kan er dan zo uitzien:

```
quicksort(List,Sorted) :- quicksort(List,Sorted,[]).
```