@languagegerman ritefiletoc@languagegerman ritefilelof@languagegerman ritefilelot@languagegerman ritefiletoc

SML-Kurs, SS 2003, Lösungen zu ausgewählten Aufgaben

Hans Leiß

7. Mai 2003

Lösung zu Aufgabe 1.1: Wir benutzen die allgemeinere Definition von aussagenlogischen Formeln, bei der Konjunktion und Disjunktion auf eine Liste von Formeln angewendet werden:

⟨*prop*⟩≡
```
datatype 'a prop =
    Var of 'a
  | Neg of 'a prop
  | Disj of 'a prop list
  | Conj of 'a prop list
```

(i) Wir können bei Disjunktion und Konjunktion die Auswertung rekursiv über die Länge der Argumentlisten definieren:

⟨*eval*⟩≡
```
fun eval h (Var a) = h a
  | eval h (Neg phi) = not (eval h phi)
  | eval h (Disj []) = false
  | eval h (Disj (phi::phis)) = (eval h phi) orelse (eval h (Disj phis))
  | eval h (Conj []) = true
  | eval h (Conj (phi::phis)) = (eval h phi) andalso (eval h (Conj phis))
```

(ii) Wir können auch etwas anders vorgehen. Bei der Auswertung einer Disjunktion von mehreren Formeln werten wir zuerst die Teilformeln aus; dadurch erhalten wir eine Liste von boole'schen Werten, die mit dem logischen *oder* auf„summiert" werden muß. Dazu ist die Faltung nützlich:

$$b_1 \vee b_2 \vee \cdots \vee b_n \;=\; b_1 \vee (b_2 \vee \cdots \vee (b_n \vee \texttt{ false})\cdots) = \texttt{foldr} \;\vee\; \texttt{false}\; [b_1, \ldots, b_n]$$

Da $\vee$ bzw. das `orelse` von SML nur eine eingebaute Abkürzung, aber kein Wert ist, müssen wir damit erst eine Funktion `(fn (b,c) => (b orelse c))` bilden, die dann über die Liste der Werte der Disjunktionsglieder gefaltet wird. In SML sieht das so aus:

⟨*eval-2*⟩≡
```
fun eval h (Var a) = h a
  | eval h (Neg phi) = not (eval h phi)
  | eval h (Disj phis) =
    foldr (fn (b,c) => (b orelse c)) false (map (eval h) phis)
  | eval h (Conj phis) =
    foldr (fn (b,c) => (b andalso c)) true (map (eval h) phis)
```

Hier haben wir anonyme Funktionen für die Konjunktion bzw. Disjunktion von boole'schen Werten benutzt. Man kann diesen Funktionen auch Namen geben und sie *vor* der Definition von `eval` einführen:

$\langle$*eval-2a*$\rangle\equiv$
```
fun disj (b,c) = b orelse c
fun conj (b,c) = b andalso c
fun eval h (Var a) = h a
  | eval h (Neg phi) = not (eval h phi)
  | eval h (Disj phis) = foldr disj false (map (eval h) phis)
  | eval h (Conj phis) = foldr conj true (map (eval h) phis)
```

Man kann aber auch drei Funktionen `eval,conj,disj` *simultan* deklarieren. Dazu vebindet man einzelne Deklarationen mit `and`:

$\langle$*eval-2b*$\rangle\equiv$
```
fun eval h (Var a) = h a
  | eval h (Neg phi) = not (eval h phi)
  | eval h (Disj phis) = foldr disj false (map (eval h) phis)
  | eval h (Conj phis) = foldr conj true (map (eval h) phis)
and disj (b,c) = b orelse c
and conj (b,c) = b andalso c
```

In diesem Fall wird die Reihenfolge der Teildeklarationen ignoriert; daher kann man z.B. zwei Funktionen definieren, die Hilfsfunktionen für einander sind, wie in

$\langle$*expl*$\rangle\equiv$
```
fun even x = if x > 0 then (odd (x-1)) else if x < 0 then (odd (x+1)) else true
and odd x = if x > 0 then (even (x-1)) else if x < 0 then (even (x+1)) else false
```

△

Lösung zu Aufgabe 1.2: In der Definition war der Fall „$nnf(\neg\neg\varphi) := \varphi$, für φ atomar“, nicht korrekt: wir dürfen uns hier nicht auf atomares φ beschränken. Ist aber φ nicht atomar, so kann es selbst wieder Negationen enthalten. Daher muß die Klausel korrekt so heißen: „$nnf(\neg\neg\varphi) := nnf(\varphi)$“.

Eine mögliche Lösung lautet daher:

$\langle$*nnf*$\rangle\equiv$
```
fun nnf (Var a) = Var a
  | nnf (Conj phis) = Conj (map nnf phis)
  | nnf (Disj phis) = Disj (map nnf phis)
  | nnf (Neg (Var a)) = Neg (Var a)
  | nnf (Neg (Conj phis)) = Disj (map (nnf o Neg) phis)
  | nnf (Neg (Disj phis)) = Conj (map (nnf o Neg) phis)
  | nnf (Neg (Neg phi)) = nnf phi
```

Die Funktion `nnf o Neg` ist die Komposition der Funktionen `nnf` und `Neg`. Beachte, daß Wertkonstruktoren wie `Neg` für SML *Funktionen* sind:

$\langle$*test*$\rangle\equiv$
```
- Neg;
val it = fn : 'a prop -> 'a prop
```

Die *Komposition* `(f o g)` der (einstelligen) Funktionen `f` und `g` ist in SML vorhanden; man könnte sie definieren durch:

$\langle$*compose*$\rangle\equiv$
```
infix comp;
fun (f comp g) = fn x => f(g x)
```

Nach der Infix-Erklärung müs die Infixschreibweise schon in der Deklaration der Funktion benutzt werden.

Ist z.B. `comp` eine (2-stellige) Funktion, für die Infixschreibweise erklärt wurde, so kann man wie in `(op comp) (f,h)` auch die Präfixschreibweise benutzen; ebenso ist `(op comp)` nützlich, wenn man die Funktion als Argument eines Funktionals (wie `map`) benutzen will. △

Lösung zu Aufgabe 1.3:

(i) Nimmt man eine Zuordnung `s : 'a -> 'a prop` zur Ersetzung atomarer Formlen `Var n` durch `s n` an, so kann man die Substitution definieren durch:

$\langle$*subst-a*$\rangle\equiv$

```
fun subst s (Var n) = s n
  | subst s (Neg phi) = (Neg (subst s phi))
  | subst s (Conj phis) = (Conj (map (subst s) phis))
  | subst s (Disj phis) = (Disj (map (subst s) phis))
```

(ii) Nimmt man eine Zuordnung `s : ('a * 'a prop)` an, so muß man bei atomaren Formeln zuerst die Liste durchsuchen:

$\langle$*subst-b*$\rangle\equiv$

```
exception Undefined;
fun subst ((m,phi)::rest) (Var n) = if m=n then phi else subst rest (Var n)
  | subst [] (Var n) = raise Undefined
  | subst s (Neg phi) = (Neg (subst s phi))
  | subst s (Conj phis) = (Conj (map (subst s) phis))
  | subst s (Disj phis) = (Disj (map (subst s) phis))
```

Hier wird zuerst für den Fall, daß man für ein `Var n` kein passendes Paar `(n,phi)` in der Liste `s` findet, eine Ausnahme erklärt.

Die Definition und Behandlung von Ausnahmen müssen wir uns später einmal genauer ansehen.

△