

Diskussion:

- `for y=... to ... do ... done` ist eine vereinfachte `while`-Schleife ;-)
- Für die Initialisierung des Arrays `count` iterieren wir einmal über alle Knoten und für jeden Knoten einmal über alle ausgehenden Kanten.

⇒ Aufwand: proportional $n + m$

- Die Hauptschleife behandelt jeden Knoten maximal einmal.
- `dfs` wird maximal einmal für jeden Knoten und jede Kante aufgerufen und behandelt jede Kante maximal einmal.

⇒ Aufwand: proportional $n + m$

⇒ Der Gesamtaufwand ist linear :-)

6 Formale Methoden für Ocaml

Frage:

Wie können wir uns versichern, dass ein Ocaml-Programm das macht, was es tun soll ???

Wir benötigen:

- eine formale Semantik;
- Techniken, um Aussagen über Programme zu beweisen ...

6.1 MiniOcaml

Um uns das Leben leicht zu machen, betrachten wir nur einen kleinen Ausschnitt aus Ocaml. Wir erlauben ...

- nur die Basistypen `int`, `bool` sowie Tupel und Listen;
- rekursive Funktionsdefinitionen nur auf dem `Top-Level` :-)

Wir verbieten ...

- veränderbare Datenstrukturen;
- Ein- und Ausgabe;
- lokale rekursive Funktionen :-)

Dieses Fragment von **Ocaml** nennen wir **MiniOcaml**.

Ausdrücke in **MiniOcaml** lassen sich durch die folgende Grammatik beschreiben:

$$\begin{aligned} E ::= & \text{const} \mid \text{name} \mid \text{op}_1 E \mid E_1 \text{op}_2 E_2 \mid \\ & (E_1, \dots, E_k) \mid \text{let name} = E_1 \text{ in } E_0 \mid \\ & \text{match } E \text{ with } P_1 \rightarrow E_1 \mid \dots \mid P_k \rightarrow E_k \mid \\ & \text{fun name} \rightarrow E \mid E E_1 \end{aligned}$$
$$P ::= \text{const} \mid \text{name} \mid (P_1, \dots, P_k) \mid P_1 :: P_2$$

Dieses Fragment von **Ocaml** nennen wir **MiniOcaml**.

Ausdrücke in **MiniOcaml** lassen sich durch die folgende Grammatik beschreiben:

$$\begin{aligned} E ::= & \text{const} \mid \text{name} \mid \text{op}_1 E \mid E_1 \text{op}_2 E_2 \mid \\ & (E_1, \dots, E_k) \mid \text{let name} = E_1 \text{ in } E_0 \mid \\ & \text{match } E \text{ with } P_1 \rightarrow E_1 \mid \dots \mid P_k \rightarrow E_k \mid \\ & \text{fun name} \rightarrow E \mid E E_1 \end{aligned}$$
$$P ::= \text{const} \mid \text{name} \mid (P_1, \dots, P_k) \mid P_1 :: P_2$$

Abkürzung:

$$\text{fun } x_1 \rightarrow \dots \text{fun } x_k \rightarrow e \equiv \text{fun } x_1 \dots x_k \rightarrow e$$

Achtung:

- Die Menge der **erlaubten** Ausdrücke muss weiter eingeschränkt werden auf diejenigen, die **typkorrekt** sind, d.h. für die der **Ocaml**-Compiler einen Typ herleiten kann ...
 - (1, [true; false]) **typkorrekt**
 - (1 [true; false]) nicht **typkorrekt**
 - ([1; true], false) nicht **typkorrekt**
- Wir verzichten auf `if ... then ... else ...`, da diese durch `match ... with true -> ... | false -> ...` simuliert werden können :-)
- Wir hätten auch auf `let ... in ...` verzichten können (wie?)

Ein **Programm** besteht dann aus einer Folge wechselseitig rekursiver globaler Definitionen von Variablen f_1, \dots, f_m :

```
let rec  $f_1 = E_1$   
      and  $f_2 = E_2$   
      ...  
      and  $f_m = E_m$ 
```

6.2 Eine Semantik für MiniOcaml

Frage:

Zu welchem Wert wertet sich ein Ausdruck E aus ??

Ein Wert ist ein Ausdruck, der nicht weiter ausgerechnet werden kann :-)

Die Menge der Werte lässt sich ebenfalls mit einer Grammatik beschreiben:

$$V ::= \text{const} \mid \text{fun name}_1 \dots \text{name}_k \rightarrow E \mid \\ (V_1, \dots, V_k) \mid [] \mid V_1 :: V_2$$

Ein MiniOcaml-Programm ...

```
let rec comp = fun f g x -> f (g x)
    and map   = fun f list -> match list
                        with [] -> []
                        | x::xs -> f x :: map f xs
```

Ein MiniOcaml-Programm ...

```
let rec comp = fun f g x -> f (g x)
    and map   = fun f list -> match list
        with [] -> []
             | x::xs -> f x :: map f xs
```

Beispiele für Werte ...

```
1
(1, [true; false])
fun x -> 1 + 1
[fun x -> x+1; fun x -> x+2; fun x -> x+3]
```

Idee:

- Wir definieren eine Relation: $e \Rightarrow v$ zwischen Ausdrücken und ihren Werten \implies **Big-Step operationelle Semantik**.
- Diese Relation definieren wir mit Hilfe von Axiomen und Regeln, die sich an der **Struktur** von e orientieren **:-)**
- Offenbar gilt stets: $v \Rightarrow v$ für jeden Wert v **:-))**

Tupel:

$$\frac{e_1 \Rightarrow v_1 \quad \dots \quad e_k \Rightarrow v_k}{(e_1, \dots, e_k) \Rightarrow (v_1, \dots, v_k)}$$

Listen:

$$\frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2}{e_1 :: e_2 \Rightarrow v_1 :: v_2}$$

Globale Definitionen:

$$\frac{f = e \quad e \Rightarrow v}{f \Rightarrow v}$$

Lokale Definitionen:

$$\frac{e_1 \Rightarrow v_1 \quad e_0[v_1/x] \Rightarrow v_0}{\text{let } x = e_1 \text{ in } e_0 \Rightarrow v_0}$$

Funktionsaufrufe:

$$\frac{e \Rightarrow \text{fun } x \rightarrow e_0 \quad e_1 \Rightarrow v_1 \quad e_0[v_1/x] \Rightarrow v_0}{e \ e_1 \Rightarrow v_0}$$

Durch mehrfache Anwendung der Regel für Funktionsaufrufe können wir zusätzlich eine Regel für Funktionen mit **mehreren** Argumenten ableiten:

$$\frac{e \Rightarrow \text{fun } x_1 \dots x_k \rightarrow e_0 \quad e_1 \Rightarrow v_1 \dots e_k \Rightarrow v_k \quad e_0[v_1/x_1, \dots, v_k/x_k] \Rightarrow v_0}{e \ e_1 \dots e_k \Rightarrow v_0}$$

Diese abgeleitete Regel macht Beweise etwas weniger umständlich :-)

Pattern Matching:

$$\frac{e_0 \Rightarrow v' \equiv p_i[v_1/x_1, \dots, v_k/x_k] \quad e_i[v_1/x_1, \dots, v_k/x_k] \Rightarrow v}{\text{match } e_0 \text{ with } p_1 \rightarrow e_1 \mid \dots \mid p_m \rightarrow e_m \Rightarrow v}$$

— sofern v' auf keines der Muster p_1, \dots, p_{i-1} passt ;-)

Eingebaute Operatoren:

$$\frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2 \quad v_1 \text{ op } v_2 = v}{e_1 \text{ op } e_2 \Rightarrow v}$$

Die unären Operatoren behandeln wir analog :-)

Beispiel:

```
let f = fun x -> x+1
let s = fun y -> y*y
```

 $f = \text{fun } x \rightarrow x+1$ $16+1 = 17$

 $s = \text{fun } y \rightarrow y*y$ $2*2 = 4$

 $f \Rightarrow \text{fun } x \rightarrow x+1$ $16+1 \Rightarrow 17$

 $s \Rightarrow \text{fun } y \rightarrow y*y$ $2*2 \Rightarrow 4$

 $f \ 16 \Rightarrow 17$ $s \ 2 \Rightarrow 4$ $17+4 = 21$

 $f \ 16 + s \ 2 \Rightarrow 21$

// Benutzungen von $v \Rightarrow v$ haben wir i.a. weggelassen :-)

Beispiel:

```
let rec app = fun x y -> match x
  with [] -> y
       | h::t -> h :: app t y
```

Behauptung: $\text{app } (1::[]) (2::[]) \Rightarrow 1::2::[]$

Beweis:

$$\frac{\frac{\frac{\text{app} = \text{fun } x \text{ } y \text{ } \rightarrow \dots}{\text{app} \Rightarrow \text{fun } x \text{ } y \text{ } \rightarrow \dots}}{\text{app} \Rightarrow \text{fun } x \text{ } y \text{ } \rightarrow \dots} \quad \frac{\frac{\frac{\text{app} = \text{fun } x \text{ } y \text{ } \rightarrow \dots}{\text{app} \Rightarrow \text{fun } x \text{ } y \text{ } \rightarrow \dots} \quad \frac{2::[] \Rightarrow 2::[]}{\text{match } [] \dots \Rightarrow 2::[]}}{\text{app } [] \text{ } (2::[]) \Rightarrow 2::[]}}{\text{1} :: \text{app } [] \text{ } (2::[]) \Rightarrow 1::2::[]}}{\text{match } 1::[] \dots \Rightarrow 1::2::[]}}{\text{app } (1::[]) \text{ } (2::[]) \Rightarrow 1::2::[]}$$

// Benutzungen von $v \Rightarrow v$ haben wir i.a. weggelassen :-)

