5.5 Getrennte Übersetzung

- Eigentlich möchte man Ocaml-Programme nicht immer in der interaktiven Umgebung starten :-)
- Dazu gibt es u.a. den Compiler `ocamlc` ...

```
> ocamlc Test.ml
```

  interpretiert den Inhalt der Datei `Test.ml` als Folge von Definitionen einer Struktur `Test`.
- Als Ergebnis der Übersetzung liefert `ocamlc` die Dateien:

`Test.cmo`	Bytecode für die Struktur
`Test.cmi`	Bytecode für das Interface
`a.out`	lauffähiges Programm

- Gibt es eine Datei `Test.mli` wird diese als Definition der Signatur für `Test` aufgefasst. Dann rufen wir auf:

```
> ocamlc Test.mli Test.ml
```

- Benutzt eine Struktur `A` eine Struktur `B`, dann sollte diese mit übersetzt werden:

```
> ocamlc B.mli B.ml A.mli A.ml
```

- Möchte man auf die Neuübersetzung von `B` verzichten, kann man `ocamlc` auch die vor-übersetzte Datei mitgeben:

```
> ocamlc B.cmo A.mli A.ml
```

- Zur praktischen Verwaltung von benötigten Neuübersetzungen nach Änderungen von Dateien bietet Linux das Kommando `make` an. Das Protokoll der auszuführenden Aktionen steht dann in einer Datei `Makefile` :-)

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:

$$
\begin{array}{lll}
E & ::= & \texttt{const} \mid \texttt{name} \mid \texttt{op}_1\, E \mid E_1\, \texttt{op}_2\, E_2 \mid \\
& & (E_1, \ldots, E_k) \mid \texttt{let name} = E_1 \texttt{ in } E_0 \mid \\
& & \texttt{match } E \texttt{ with } P_1 \texttt{ -> } E_1 \texttt{ | } \ldots \texttt{ | } P_k \texttt{ -> } E_k \mid \\
& & \texttt{fun name -> } E \mid E\ E_1
\end{array}
$$

$$
P ::= \texttt{const} \mid \texttt{name} \mid (P_1, \ldots, P_k) \mid P_1 \texttt{ :: } P_2
$$

Dieses Fragment von Ocaml nennen wir MiniOcaml.

Ausdrücke in MiniOcaml lassen sich durch die folgende Grammatik beschreiben:

$$
\begin{array}{lll}
E & ::= & \mathsf{const} \mid \mathsf{name} \mid \mathsf{op}_1\, E \mid E_1\, \mathsf{op}_2\, E_2 \mid \\
& & (E_1, \ldots, E_k) \mid \texttt{let name = } E_1 \texttt{ in } E_0 \mid \\
& & \texttt{match } E \texttt{ with } P_1 \texttt{ -> } E_1 \texttt{ | } \ldots \texttt{ | } P_k \texttt{ -> } E_k \mid \\
& & \texttt{fun name -> } E \mid E\ E_1
\end{array}
$$

$$
P ::= \mathsf{const} \mid \mathsf{name} \mid (P_1, \ldots, P_k) \mid P_1 :: P_2
$$

Abkürzung:

$$
\texttt{fun } x_1 \texttt{ -> } \ldots \texttt{fun } x_k \texttt{ -> } e \quad \equiv \quad \texttt{fun } x_1 \ \ldots\ x_k \texttt{ -> } e
$$

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 $f_1, \ldots, f_m$:

$$
\begin{array}{lll}
\texttt{let rec} & f_1 & = \quad E_1 \\
\texttt{and} & f_2 & = \quad E_2 \\
& \ldots & \\
\texttt{and} & f_m & = \quad E_m
\end{array}
$$

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:

$$V \quad ::= \quad \text{const} \quad | \quad \texttt{fun}\ \text{name}_1\ \ldots\ \text{name}_k\ \texttt{->}\ E \quad |$$
$$(V_1, \ldots, V_k) \quad | \quad \texttt{[]} \quad | \quad V_1 :: V_2$$

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 $\Longrightarrow$ 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:

$$\frac{e_1 \Rightarrow v_1 \qquad \ldots \qquad e_k \Rightarrow v_k}{(e_1, \ldots, e_k) \Rightarrow (v_1, \ldots, v_k)}$$

Listen:

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

Globale Definitionen:

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

Lokale Definitionen:

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

Funktionsaufrufe:

$$\frac{e \Rightarrow \texttt{fun}\ x\ \texttt{->}\ e_0 \quad e_1 \Rightarrow v_1 \quad e_0[v_1/x] \Rightarrow v_0}{e\ e_1 \Rightarrow v_0}$$

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

$$\frac{e_0 \Rightarrow \texttt{fun}\ x_1 \ldots x_k \texttt{->}\, e \quad e_1 \Rightarrow v_1 \ \ldots\ e_k \Rightarrow v_k \quad e[v_1/x_1, \ldots, v_k/x_k] \Rightarrow v}{e_0\ e_1\ \ldots\ e_k \Rightarrow v}$$

Diese abgeleitete Regel macht Beweise etwas weniger umständlich :-)

Pattern Matching:

$$\frac{e_0 \Rightarrow v' \equiv p_i[v_1/x_1,\ldots,v_k/x_k] \qquad e_i[v_1/x_1,\ldots,v_k/x_k] \Rightarrow v}{\texttt{match}\ e_0\ \texttt{with}\ p_1\ \texttt{->}\ e_1\ \texttt{|}\ \ldots\ \texttt{|}\ p_m\ \texttt{->}\ e_m \Rightarrow v}$$

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

Eingebaute Operatoren:

$$\frac{e_1 \Rightarrow v_1 \qquad e_2 \Rightarrow v_2 \qquad v_1\ \mathsf{op}\ v_2 \Rightarrow v}{e_1\ \mathsf{op}\ e_2 \Rightarrow v}$$

Die unären Operatoren behandeln wir analog :-)

Der eingebaute Gleichheits-Operator:

$$v = v \quad \Rightarrow \quad \texttt{true}$$

$$v_1 = v_2 \quad \Rightarrow \quad \texttt{false}$$

sofern v, v_1, v_2 Werte sind, in denen keine Funktionen vorkommen, und v_1, v_2 syntaktisch verschieden sind :-)

Beispiel 1:

$$\frac{\texttt{17+4} \Rightarrow \texttt{21} \qquad \texttt{21} \Rightarrow \texttt{21} \qquad \texttt{21=21} \Rightarrow \texttt{true}}{\texttt{17 + 4 = 21} \Rightarrow \texttt{true}}$$

Beispiel 2:

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

$$\dfrac{\dfrac{\dfrac{}{\texttt{f = fun x -> x+1}}}{\texttt{f} \Rightarrow \texttt{fun x -> x+1} \qquad \texttt{16+1} \Rightarrow \texttt{17}}}{\texttt{f 16} \Rightarrow \texttt{17}} \qquad \dfrac{\dfrac{\dfrac{}{\texttt{s = fun y -> y*y}}}{\texttt{s} \Rightarrow \texttt{fun y -> y*y} \qquad \texttt{2*2} \Rightarrow \texttt{4}}}{\texttt{s 2} \Rightarrow \texttt{4}} \qquad \texttt{17+4} \Rightarrow \texttt{21}$$

$$\texttt{f 16 + s 2} \Rightarrow \texttt{21}$$

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

Beispiel 3:

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

Behauptung: `app (1::[]) (2::[])` $\Rightarrow$ `1::2::[]`

Beweis:

$$\dfrac{\dfrac{\texttt{app = fun x y -> ...}}{\texttt{app} \Rightarrow \texttt{fun x y -> ...}} \quad \dfrac{\texttt{2::[]} \Rightarrow \texttt{2::[]}}{\texttt{match [] ... } \Rightarrow \texttt{2::[]}}}{\dfrac{\texttt{app [] (2::[])} \Rightarrow \texttt{2::[]}}{\texttt{1 :: app [] (2::[])} \Rightarrow \texttt{1::2::[]}}} \quad \dfrac{\texttt{app = fun x y -> ...}}{\texttt{app} \Rightarrow \texttt{fun x y -> ...}}}{\dfrac{}{\texttt{match 1::[] ...} \Rightarrow \texttt{1::2::[]}}}$$

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

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

Diskussion:

- Die Big-Step operationelle Semantik ist nicht sehr gut geeignet, um Schritt für Schritt nachzu vollziehen, was ein MiniOcaml-Programm macht :-(
- Wir können damit aber sehr gut nachweisen, dass die Auswertung eine Funktion für bestimmte Argumentwerte stets terminiert:

  Dazu muss nur nachgewiesen werden, dass es jeweils einen Wert gibt, zu dem die entsprechende Funktionsanwendung ausgewertet werden kann ...