7 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 ...

7.1 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:

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

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

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 $f_1, \ldots, f_m$:

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

7.2 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:

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

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
```

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

// 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:

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

// 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 ...

Beispiel-Behauptung:

app l_1 l_2 terminiert für alle Listen-Werte l_1, l_2.

Beweis:

Induktion nach der Länge n der Liste l_1.

$n = 0$: D.h. l_1 = `[]`. Dann gilt:

$$\dfrac{\dfrac{\texttt{app = fun x y -> } \cdots}{\texttt{app} \Rightarrow \texttt{fun x y -> } \cdots} \qquad \texttt{match [] with [] -> } l_2 \texttt{ | ...} \Rightarrow l_2}{\texttt{app [] } l_2 \Rightarrow l_2}$$

:-)

$n > 0$: D.h. l_1 = `h::t`.

Insbesondere nehmen wir an, dass die Behauptung bereits für alle kürzeren Listen gilt. Deshalb haben wir:

$$\texttt{app t } l_2 \Rightarrow l$$

für ein geeignetes l. Wir schließen:

$$\dfrac{\dfrac{\texttt{app = fun x y -> ...}}{\texttt{app} \Rightarrow \texttt{fun x y -> ...}} \quad \dfrac{\dfrac{\texttt{app t } l_2 \Rightarrow l}{\texttt{h :: app t } l_2 \Rightarrow \texttt{h :: } l}}{\texttt{match h::t with } \cdots \Rightarrow \texttt{h :: } l}}{\texttt{app (h::t) } l_2 \Rightarrow \texttt{h :: } l}$$

:-)

Diskussion (Forts.):

- Wir können mit der Big-step-Semantik auch überprüfen, dass optimierende Transformationen korrekt sind :-)
- Schließlich können wir sie benutzen, um die Korrektheit von Aussagen über funktionale Programme zu beweisen !
- Die Big-Step operationelle Semantik legt dabei nahe, Ausdrücke als Beschreibungen von Werten aufzufassen.
- Ausdrücke, die sich zu den gleichen Werten auswerten, sollten deshalb austauschbar sein ...

Achtung:

- Gleichheit zwischen Werten kann in MiniOcaml nur getestet werden, wenn diese keine Funktionen enthalten !!
- Solche Werte nennen wir vergleichbar. Sie haben die Form:

$$C \quad ::= \quad \mathsf{const} \quad | \quad (C_1, \ldots, C_k) \quad | \quad [] \quad | \quad C_1 :: C_2$$

Achtung:

- Gleichheit zwischen Werten kann in MiniOcaml nur getestet werden, wenn diese keine Funktionen enthalten !!
- Solche Werte nennen wir vergleichbar. Sie haben die Form:

$$C \quad ::= \quad \text{const} \quad | \quad (C_1, \ldots, C_k) \quad | \quad \texttt{[]} \quad | \quad C_1 \texttt{::} C_2$$

- Offenbar ist ein MiniOcaml-Wert genau dann vergleichbar, wenn sein Typ funktionsfrei, d.h. einer der folgenden Typen ist:

$$c \quad ::= \quad \texttt{bool} \quad | \quad \texttt{int} \quad | \quad \texttt{unit} \quad | \quad c_1 * \ldots * c_k \quad | \quad c\ \texttt{list}$$

:-)

Diskussion

- In Programmoptimierungen möchten wir gelegentlich Funktionen austauschen, z.B.

```
comp (map f) (map g) = map (comp f g)
```

- Offenbar stehen rechts und links des Gleichheitszeichens Funktionen, deren Gleichheit Ocaml nicht überprüfen kann

$\Longrightarrow$

Die Logik benötigt einen stärkeren Gleichheitsbegriff :-)

Erweiterung der Gleichheit:

Wir erweitern die Ocaml-Gleichheit = auf Werten auf Ausdrücke, die nicht terminieren, und Funktionen.

Nichtterminierung:

$$\frac{e_1, e_2 \qquad \text{terminieren beide nicht}}{e_1 = e_2}$$

Terminierung:

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

Strukturierte Werte:

$$\frac{v_1 = v_1' \quad \ldots \quad v_k = v_k'}{(v_1, \ldots, v_k) = (v_1', \ldots, v_k')}$$

$$\frac{v_1 = v_1' \qquad v_2 = v_k'}{v_1 :: v_2 = v_1' :: v_2'}$$

Funktionen:

$$\frac{e_1[v/x_1] = e_2[v/x_2] \quad \text{für alle} \quad v}{\texttt{fun}\ x_1\ \texttt{->}\ e_1 = \texttt{fun}\ x_2\ \texttt{->}\ e_2}$$

$\Longrightarrow$ extensionale Gleichheit

Wir haben:

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

Seien der Typ von e_1, e_2 funktionsfrei. Dann gilt:

$$\frac{e_1 = e_2 \qquad\qquad e_1 \quad \text{terminiert}}{e_1 = e_2 \Rightarrow \texttt{true}}$$

Das entscheidende Hilfsmittel für unsere Beweise ist das ...

Substitutionslemma:

$$\frac{e_1 = e_2}{e[e_1/x] = e[e_2/x]}$$

Wir folgern für funktionsfreie Ausdrücke e_1, e_2, e:

$$\frac{e_1 = e_2 \Rightarrow \texttt{true} \qquad e[e_1/x] \text{ terminiert}}{e[e_1/x] = e[e_2/x] \Rightarrow \texttt{true}}$$

Diskussion:

- Das Lemma besagt damit, dass wir in jedem Kontext alle Vorkommen eines Ausdrucks e_1 durch einen Ausdruck e_2 ersetzen können, sofern e_1 und e_2 die selben Werte representieren :-)
- Das Lemma lässt sich mit Induktion über die Tiefe der benötigten Herleitungen zeigen (was wir uns sparen :-))
- Der Austausch von als gleich erwiesenen Ausdrücken gestattet uns, die Äquivalenz von Ausdrücken zu beweisen ...

Zuerst verschaffen wir uns ein Repertoir von Umformungsregeln, die die Gleichheit von Ausdrücken auf Gleichheiten anderer, möglicherweise einfacherer Ausdrücke zurück führt ...

Vereinfachung lokaler Definitionen:

$$\frac{e_1 \text{ terminiert}}{\texttt{let } x = e_1 \texttt{ in } e \quad = \quad e[e_1/x]}$$

Zuerst verschaffen wir uns ein Repertoir von Umformungsregeln, die die Gleichheit von Ausdrücken auf Gleichheiten anderer, möglicherweise einfacherer Ausdrücke zurück führt ...

Vereinfachung lokaler Definitionen:

$$\frac{e_1 \text{ terminiert}}{\texttt{let } x = e_1 \texttt{ in } e \quad = \quad e[e_1/x]}$$

Vereinfachung von Funktionsaufrufen:

$$\frac{e_0 \quad = \quad \texttt{fun } x \texttt{ -> } e \qquad e_1 \text{ terminiert}}{e_0\ e_1 \quad = \quad e[e_1/x]}$$

Beweis der let-Regel:

Weil e_1 terminiert, gibt es einen Wert v_1 mit:

$$e_1 \Rightarrow v_1$$

Wegen des Substitutionslemmas gilt dann auch:

$$e[v_1/x] = e[e_1/x]$$

Fall 1: $e[v_1/x]$ terminiert.

Dann gibt es einen Wert v mit:

$$e[v_1/x] \Rightarrow v$$

Deshalb haben wir:

$$e[e_1/x] = e[v_1/x] = v$$

Wegen der Big-step operationellen Semantik gilt dann aber:

$$\texttt{let } x = e_1 \texttt{ in } e \quad \Rightarrow \quad v \qquad \text{und damit:}$$
$$\texttt{let } x = e_1 \texttt{ in } e \quad = \quad e[e_1/x]$$

Fall 2: $e[v_1/x]$ terminiert nicht.

Dann terminiert $e[e_1/x]$ nicht und auch nicht $\texttt{let } x = e_1 \texttt{ in } e$.

Folglich gilt:

$$\texttt{let } x = e_1 \texttt{ in } e \; = \; e[e_1/x]$$

:-)