6.5 Getrennte Übersetzung

- Eigentlich möchte man Ocaml-Programme nicht immer in der interaktiven Umgebung starten.
- Dazu gibt es u.a. den Compiler `ocamlc ...
  > ocamlc Test.mli
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`.
- ... oder man benutzt gleich `ocamlbuild`. 
7 Formale Methoden für Ocaml

Frage

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

Wir benötigen:
- eine formaleSemantik;
- Techniken, um Aussagen über Programme zu beweisen ...

Dieses Fragment von Ocaml nennen wir MiniOcaml.

Ausdrücke in MiniOcaml lassen sich durch die folgende Grammatik beschreiben:

\[
E ::= \text{const} \mid \text{name} \mid \text{op}_1 E \mid E_1 \text{ op}_2 E_2 \mid (E_1, \ldots, E_k) \mid \text{let } \text{name} = E_1 \text{ in } E_0 \mid \text{match } E \text{ with } P_1 \rightarrow E_1 \mid \ldots \mid P_k \rightarrow E_k \mid \text{fun } \text{name} \rightarrow E \mid E \text{ } E_1
\]

\[
P ::= \text{const} \mid \text{name} \mid (P_1, \ldots, P_j) \mid P_1 :: P_2
\]

Abkürzung

\[
\text{fun } x_1 \rightarrow \ldots \text{fun } x_k \rightarrow \text{ } \equiv \text{ fun } x_1 \ldots x_k \rightarrow \epsilon
\]

7.1 MiniOcaml

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

- nur die Basistypen \text{int}, \text{bool} sowie Tupel und Listen;
- rekursive Funktionsdefinitionen nur auf dem Top-Level;

Wir verbieten ...

- veränderbare Datenstrukturen;
- Ein- und Ausgabe;
- locale rekursive Funktionen;

Dieses Fragment von Ocaml nennen wir MiniOcaml.

Ausdrücke in MiniOcaml lassen sich durch die folgende Grammatik beschreiben:

\[
E ::= \text{const} \mid \text{name} \mid \text{op}_1 E \mid E_1 \text{ op}_2 E_2 \mid (E_1, \ldots, E_k) \mid \text{let } \text{name} = E_1 \text{ in } E_0 \mid \text{match } E \text{ with } P_1 \rightarrow E_1 \mid \ldots \mid P_k \rightarrow E_k \mid \text{fun } \text{name} \rightarrow E \mid E \text{ } E_1
\]

\[
P ::= \text{const} \mid \text{name} \mid (P_1, \ldots, P_j) \mid P_1 :: P_2
\]

Abkürzung

\[
\text{fun } x_1 \rightarrow \ldots \text{fun } x_k \rightarrow \text{ } \equiv \text{ fun } x_1 \ldots x_k \rightarrow \epsilon
\]
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 \( f_1, \ldots, f_m \):

\[
\text{let rec } f_1 = E_1 \\
\quad \text{and } f_2 = E_2 \\
\quad \ldots \\
\quad \text{and } f_m = E_m
\]

Dieses Fragment von Ocaml nennen wir MiniOcaml.
Ausdrücke in MiniOcaml lassen sich durch die folgende Grammatik beschreiben:

\[
E :: = \text{const} \mid \text{name} \mid \text{op}_1 E \mid \text{E}_1 \text{op}_2 E_2 \\
\quad \text{(E}_1, \ldots, E_k) \mid \text{let name } = E_2 \text{ in } E_0 \\
\quad \text{match } E \text{ with } P_1 \rightarrow E_1 \mid \ldots \mid P_k \rightarrow E_k \\
\quad \text{fun name } \rightarrow E \mid E E_1
\]

\[
P :: = \text{const} \mid \text{name} \mid (P_1, \ldots, P_k) \mid P_1 :: P_2
\]

Abkürzung

\[
\text{fun } x_1 \rightarrow \ldots \text{fun } x_k \rightarrow e \equiv \text{fun } x_1 \ldots x_k \rightarrow e
\]

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:

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

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 \( \Rightarrow \) 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

\((e_1 \Rightarrow v_1) \quad \ldots \quad (e_k \Rightarrow v_k)\) \( \Rightarrow (v_1, \ldots, v_k)\)

Listen

\( e_1 \Rightarrow v_1 \Rightarrow v_2 \Rightarrow v \Rightarrow v \)
\( e_1 \Rightarrow v_1 \Rightarrow v_1 \Rightarrow v_2 \Rightarrow v_2 \)

Globale Definitionen

\[ f = e \quad e \Rightarrow v \]
\[ f \Rightarrow v \]
**Lokale Definitionen**

\[ v_1 \Rightarrow v_1 \]
\[ e_0[v_1/x] \Rightarrow v_0 \]

let \( x = e_1 \) in \( e_0 \) \( \Rightarrow v_0 \)

**Funktionsaufrufe**

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

\[ e \ e_1 \Rightarrow v_0 \]

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

\[ e_0 \Rightarrow \text{fun} \ x_1 \ldots x_k : \rightarrow e \]
\[ e_1 \Rightarrow v_1 \]
\[ e_2 \Rightarrow v_2 \]
\[ \ldots \]
\[ e_k \Rightarrow v_k \]

\[ e[v_1/x_1, \ldots, v_k/x_k] \Rightarrow v \]

\[ e_0 \ e_1 \ \ldots \ e_k \Rightarrow v \]

Diese abgeleitete Regel macht Beweise etwas weniger umständlich.

**Pattern Matching**

\[ e_0 \Rightarrow v' \| p_1[v_1/x_1, \ldots, v_k/x_k] \]
\[ p_1[v_1/x_1, \ldots, v_k/x_k] \Rightarrow v \]

match \( e_0 \) with \( p_1 \rightarrow e_1 \ | \ \ldots \ | \ p_m \rightarrow e_m \Rightarrow v \)

— sofern \( v' \) auf keines der Muster \( p_1, \ldots, p_{m-1} \) passt.

**Eingebaute Operatoren**

\[ e_1 \Rightarrow v_1 \]
\[ e_2 \Rightarrow v_2 \]
\[ v_1 \ op \ v_2 \Rightarrow v \]

\[ e_1 \ op \ e_2 \Rightarrow v \]

Die unären Operatoren behandeln wir analog.
Pattern Matching

\[ e_0 \Rightarrow v' \equiv p_i[v_1/x_1,\ldots,v_k/x_k] \quad e_j[v_1/x_1,\ldots,v_k/x_k] \Rightarrow v \]
match \( e_0 \) with \( p_1 \rightarrow e_1 \mid \ldots \mid p_m \rightarrow e_m \Rightarrow \quad v \)

sofern \( v' \) auf keines der Muster \( p_1,\ldots,p_{m-1} \) passt \( \vdash \)

Eingebaute Operatoren

\[ e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2 \quad v_1 \circ v_2 \Rightarrow v \]

\[ e_1 \circ e_2 \Rightarrow v \]

Die unären Operatoren behandeln wir analog.

Der eingebaute Gleichheits-Operator

\[ v = v \Rightarrow \text{true} \]
\[ v_1 = v_2 \Rightarrow \text{false} \]

sofern \( v, v_1, v_2 \) Werte sind, in denen keine Funktionen vorkommen, und \( v_1, v_2 \) syntaktisch verschieden sind.

Beispiel 1

\[ 17+4 \Rightarrow 21 \quad 21 \Rightarrow 21 \quad 21 \cdot 21 \Rightarrow \text{true} \]
\[ 17 + 4 = 21 \Rightarrow \text{true} \]

Beispiel 2

\[ \text{let } f = \text{fun } x \rightarrow x+1 \]
\[ \text{let } s = \text{fun } y \rightarrow y*y \]

\[ f \circ \text{fun } x \rightarrow x+1 \]
\[ f \Rightarrow \text{fun } x \rightarrow x+1 \quad 16+1 \Rightarrow 17 \]
\[ s \Rightarrow \text{fun } y \rightarrow y*y \quad 2+2 \Rightarrow 4 \]

\[ f \text{ fun } x \rightarrow x+1 \]
\[ f \text{ 16} \Rightarrow 17 \]
\[ s \text{ 2} \Rightarrow 4 \quad 17+4 \Rightarrow 21 \]

\[ f \text{ 16} + s \text{ 2} \Rightarrow 21 \]

// Benutzungen von \( v \Rightarrow v \) haben wir i.a. weggelassen

Lokale Definitionen

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

Funktionsaufrufe

\[ e \Rightarrow \text{fun } x \rightarrow e_0 \quad e_1 \Rightarrow v_1 \quad e_0[v_1/x] \Rightarrow v_0 \]
\[ e \text{ e_1} \Rightarrow v_0 \]
Beispiel 2

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

f = fun x -> x+1
f 16 => 17

s = fun y -> y*y
s 2 => 4

17-4 => 21

f 16 + s 2 => 21

// Benutzungen von v => v haben wir i.a. weggelassen
```

Beispiel 3

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

Behauptung: app (1::[]) (2::[]) => 1::2::[]
```

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

Behauptung: app (1::[]) (2::[]) => 1::2::[]
```
**Beweis**


datei 1: \( \text{app} \underline{\text{fun x y -> ...}, 2;1 \Rightarrow 2;1} \)

\begin{align*}
\text{app} = \text{fun x y -> ...} & \Rightarrow \text{match} [] \text{ with } [] \Rightarrow l_2 & \ldots \Rightarrow l_2 \\
\text{app} [] l_2 & \Rightarrow l_2
\end{align*}

// Benutzungen von \( v \Rightarrow v \) haben wir i.a. weggelassen

\[ \text{app} (\text{fun x y -> ...}) \Rightarrow \text{match} [l_1] \ldots \Rightarrow [l_2] \]

**Beispiel-Behauptung**

\[ \text{app} l_1 l_2 \text{ terminiert für alle Listen-Werte } l_1, l_2. \]

**Beweis**

Induktion nach der Länge \( n \) der Liste \( l_1 \).

\[ n = 0 : \quad \text{D.h. } l_1 = []. \text{ Dann gilt:} \]

\[ \begin{align*}
\text{app} = \text{fun x y -> ...} & \Rightarrow \text{match} [] \text{ with } [] \Rightarrow l_2 & \ldots \Rightarrow l_2 \\
\text{app} [] l_2 & \Rightarrow l_2
\end{align*} \]

**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 …

\( n > 0 : \quad \text{D.h. } l_1 = h::t. \)

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

\[ \text{app} t l_2 \Rightarrow l \]

für ein geeignetes \( l \). Wir schließen:

\[ \begin{align*}
\text{app} = \text{fun x y -> ...} & \Rightarrow h :: \text{app} t l_2 \Rightarrow h :: l \\
\text{app} \Rightarrow \text{fun x y -> ...} & \Rightarrow \text{match} h :: t \text{ with } \ldots \Rightarrow h :: l \\
\text{app} (h::t) l_2 & \Rightarrow h :: l
\end{align*} \]
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:

\[
C ::= \text{const} \mid (C_1, \ldots, C_k) \mid [] \mid C_1 :: C_2
\]

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

\[
c ::= \text{bool} \mid \text{int} \mid \text{unit} \mid C_1 * \ldots * C_k \mid c\ \text{list}
\]

Diskussion

In Programmoptimierungen möchten wir gelegentlich Funktionen austauschen, z.B.
\[ f(g(x)) \quad f(g(x)) \]
\[ \text{comp (map f) (map g)} = \text{map (comp f g)} \]

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

Die Logik benötigt einen stärkeren Gleichheitsbegriff!