

Script generated by TTT

Title: Info2 (08.01.2016)

Date: Fri Jan 08 08:38:42 CET 2016

Duration: 89:49 min

Pages: 35

- 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`.

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.ml
```

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

<code>Test.cmo</code>	Bytecode für die Struktur
<code>Test.cmi</code>	Bytecode für das Interface
<code>a.out</code>	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 formale Semantik;
- Techniken, um Aussagen über Programme zu beweisen ...

310

Dieses Fragment von Ocaml nennen wir MiniOcaml.

Ausdrücke in MiniOcaml lassen sich durch die folgende Grammatik beschreiben:

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

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

312

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;

311

Dieses Fragment von Ocaml nennen wir MiniOcaml.

Ausdrücke in MiniOcaml lassen sich durch die folgende Grammatik beschreiben:

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

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

Abkürzung

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

313

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?)

314

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

```
let rec f1 = E1
    and f2 = E2
    ...
    and fm = Em
```

315

Dieses Fragment von Ocaml nennen wir MiniOcaml.

Ausdrücke in MiniOcaml lassen sich durch die folgende Grammatik beschreiben:

```
E ::= const | name | op1 E | E1 op2 E2 |
    (E1, ..., Ek) | let name = E1 in E0 |
    \ match E with P1 -> E1 | ... | Pk -> Ek |
    • fun name -> E | E E1
    (fun name -> E0) E1
P ::= const | name | (P1, ..., Pk) | P1 :: P2
```

Abkürzung

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

313

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 ::= const | fun name1 ... namek -> E |
    (V1, ..., Vk) | [] | V1 :: V2
```

316

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
```

317

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]
```

318

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 .

319

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

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

320

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

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

321

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

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

Diese abgeleitete Regel macht Beweise etwas weniger umständlich.

322

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

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

Diese abgeleitete Regel macht Beweise etwas weniger umständlich.

322

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

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

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

Die unären Operatoren behandeln wir analog.

323

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

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

Die unären Operatoren behandeln wir analog.

323

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.

fun f x → x+1

fun y y → y+1

Beispiel 1

$$\frac{17+4 \Rightarrow 21 \quad 21 \Rightarrow 21 \quad 21=21 \Rightarrow \text{true}}{17 + 4 = 21 \Rightarrow \text{true}}$$

324

Beispiel 2

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

$$\frac{\frac{f \text{ fun } x \rightarrow x+1}{f \Rightarrow \text{fun } x \rightarrow x+1} \quad \frac{s \text{ fun } y \rightarrow y*y}{s \Rightarrow \text{fun } y \rightarrow y*y} \quad 16+1 \Rightarrow 17 \quad 2*2 \Rightarrow 4}{f \ 16 \Rightarrow 17 \quad s \ 2 \Rightarrow 4 \quad 17+4 \Rightarrow 21} \quad \frac{16 \Rightarrow 16}{f \ 16 + s \ 2 \Rightarrow 21}$$

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

325

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

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

321

Beispiel 2

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

$f = \text{fun } x \rightarrow x+1$	$s = \text{fun } y \rightarrow y*y$
$f \Rightarrow \text{fun } x \rightarrow x+1$	$s \Rightarrow \text{fun } y \rightarrow y*y$
$f \ 16 \Rightarrow 17$	$s \ 2 \Rightarrow 4$
$16+1 \Rightarrow 17$	$2*2 \Rightarrow 4$
$f \ 16 + s \ 2 \Rightarrow 21$	

2 → 2

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

325

Beispiel 3

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

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

326

Beweis

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

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

327

Beispiel 3

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

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

326

Diskussion (Forts.)

$$c_1 + c_2 \Rightarrow c$$

- 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 ...

331

Achtung

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

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

332

Achtung

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

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

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

$$c ::= \text{bool} \mid \text{int} \mid \text{unit} \mid c_1 * \dots * c_k \mid c \text{ list}$$

333

Diskussion

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

$$f(gx) \quad f(g(x))$$
$$\text{comp (map f) (map g) = map (comp f g)}$$

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

⇒

Die Logik benötigt einen **stärkeren** Gleichheitsbegriff!

Q

334