

## Ein MiniOcaml-Programm ...

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

## Ein MiniOcaml-Programm ...

```
let rec comp = fun f -> fun g -> fun x -> f (g x)
    and map   = fun f -> fun 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 1:

- Wir definieren einen **Schritt** der Berechnung als eine Relation:  
 $e \rightarrow e'$  zwischen Ausdrücken.
- Eine **Berechnung** ist eine **Folge** von Berechnungsschritten  
 $\implies$  **Small-Step operationelle Semantik**
- Auf Werten ist kein weiterer Berechnungsschritt möglich, d.h.  
bei Erreichen eines Wert endet eine Berechnung **:-)**
- Einzelne Berechnungsschritte definieren wir mit Hilfe  
geeigneter Axiome und Regeln ...

//  $v, v_1, \dots$  stehen für Werte,

//  $e, e_1, \dots$  bezeichnen beliebige Ausdrücke

Tupel:

$$\frac{e_i \rightarrow e'_i}{(v_1, \dots, v_{i-1}, e_i, \dots, e_k) \rightarrow (v_1, \dots, v_{i-1}, e'_i, e_{i+1}, \dots, e_k)}$$

Listen:

$$\frac{e_1 \rightarrow e'_1}{e_1 :: e_2 \rightarrow e'_1 :: e_2}$$

$$\frac{e_2 \rightarrow e'_2}{v_1 :: e_2 \rightarrow v_1 :: e'_2}$$

Globale Definitionen:

$$\frac{f = e}{f \rightarrow e}$$

## Lokale Definitionen:

$$\frac{e_1 \rightarrow e'_1}{\text{let } x = e_1 \text{ in } e_0 \rightarrow \text{let } x = e'_1 \text{ in } e_0}$$

$$\text{let } x = v \text{ in } e_0 \rightarrow e_0[v/x]$$

## Funktionsaufrufe:

$$\frac{e_1 \rightarrow e'_1}{e_1 e_2 \rightarrow e'_1 e_2} \qquad \frac{e_2 \rightarrow e'_2}{v e_2 \rightarrow v e'_2}$$

$$(\text{fun } x \rightarrow e_0) v \rightarrow e_0[v/x]$$

## Pattern Matching:

$$\frac{e_0 \rightarrow e'_0}{\text{match } e_0 \text{ with } plist \rightarrow \text{match } e'_0 \text{ with } plist}$$

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

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

// “ $\equiv$ ” bezeichnet **syntaktische Gleichheit**

## Eingebaute Operatoren:

$$\frac{e_1 \rightarrow e'_1}{e_1 \text{ op } e_2 \rightarrow e'_1 \text{ op } e_2}$$

$$\frac{e_2 \rightarrow e'_2}{v \text{ op } e_2 \rightarrow v \text{ op } e'_2}$$

$$\frac{v_1 \text{ op } v_2 = v}{v_1 \text{ op } v_2 \rightarrow v}$$

Die unären Operatoren behandeln wir analog :-)

## Beispiel:

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

```
f 16 + s 2  → (fun x -> x+1) 16 + s 2
              → (16+1) + s 2
              → 17 + s 2
              → 17 + (fun y -> y*y) 2
              → 17 + (2*2)
              → 17 + 4
              → 21
```

## Beispiel:

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

## Frage:

Welchen Wert liefert `app (1::[]) (2::[])` ?

`app (1::[]) (2::[])`

- `(fun x -> fun y -> ...) (1::[]) (2::[])`
- `(fun y -> match ...) (2::[])`
- `match 1::[] with ...`
- `1 :: app [] (2::[])`
- `1 :: (fun x -> fun y -> ...) [] (2::[])`
- `1 :: (fun y -> match ...) (2::[])`
- `1 :: match [] with ...`
- `1 :: 2 :: []`

## Diskussion:

- Die Small-Step operationelle Semantik ist gut geeignet, um eine einzelne Berechnung nach zu vollziehen :-)
- Sie ist weniger gut geeignet, um nachzuweisen, dass verschiedene Ausdrücke den selben Wert liefern :-(

## Diskussion:

- Die Small-Step operationelle Semantik ist gut geeignet, um eine einzelne Berechnung nach zu vollziehen :-)
- Sie ist weniger gut geeignet, um nachzuweisen, dass verschiedene Ausdrücke den selben Wert liefern :-)

## Idee 2:

- Wir definieren eine Relation:  $e \Rightarrow v$  zwischen Ausdrücken und ihren Werten  $\implies$  Big-Step operationelle Semantik.
- Diese Relation definieren wir erneut 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:

$$\frac{e_1 \Rightarrow \text{fun } x \rightarrow e_0 \quad e_2 \Rightarrow v_2 \quad e_0[v_2/x] \Rightarrow v_0}{e_1 \ e_2 \Rightarrow v_0}$$

## 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
```

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

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

## Beispiel:

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

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



## 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 überprüfen, dass **optimierende Transformationen** korrekt sind :-)
- Schließlich können wir sie benutzen, um die Korrektheit von Aussagen über funktionale Programme zu beweisen !
- Zuerst müssen wir aber zeigen, dass sie zur **Small-Step operationellen Semantik äquivalent** ist ...

## Notation:

$e_1 \xrightarrow{*} e_2$  heißt  $e_1 \equiv e_2$  (textuelle Gleichheit) oder es gibt eine Folge von Berechnungsschritten:  $e_1 \rightarrow \dots \rightarrow e_2$

## Satz

Für jedes MiniOcaml-Programm gilt:

$e \xrightarrow{*} v$  genau dann wenn  $e \Rightarrow v$

//  $e$  Ausdruck,  $v$  Wert

## Diskussion:

- Der Satz lässt sich mit Hilfe von Induktion beweisen einmal nach der Länge von Folgen von Berechnungsschritten, einmal nach der Tiefe von Ableitungen der Big-Step operationellen Semantik :-)
- Die Big-Step operationelle Semantik legt nahe, Ausdrücke als **Beschreibungen** von Werten aufzufassen.
- Ausdrücke, die sich zu den **gleichen** Werten auswerten, sollten deshalb austauschbar sein ...