Script generated by TTT

Title: Seidl: Functional Programming and Verification (11.01.2019)

Date: Fri Jan 11 08:30:18 CET 2019

Duration: 90:21 min

Pages: 12

By repeated application of the rule for function calls, a rule for functions with multiple arguments can be derived:

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

This derived rule makes proofs somewhat simpler.

The built-in equality operator

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

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

given that v, v_1, v_2 are values that do not contain functions, and v_1, v_2 are syntactically different.

Example 1

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

The built-in equality operator

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

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

given that v, v_1, v_2 are values that do not contain functions, and v_1, v_2 are syntactically different.

Example 1

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

Discussion

- The big-step operational semantics is not well suited for tracking step-by-step how evaluation by MiniOcaml proceeds.
- It is quite convenient, though, for proving that the evaluation of a function for particular argument values terminates:

  For that, it suffices to prove that there are values to which the corresponding function calls can be evaluated ...

Example Claim

app l_1 l_2 terminates for all list values l_1, l_2.

Proof

Induction on the length n of the list l_1.

$n = 0$ I.e., l_1 = `[]`. Then

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

$n > 0$: I.e., l_1 = `h::t`.

In particular, we assume that the claim already holds for all shorter lists. Then we have:

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

for some l. We deduce

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

Example Claim

app l_1 l_2 terminates for all list values l_1, l_2.

Proof

Induction on the length n of the list l_1.

$n = 0$ I.e., l_1 = `[]`. Then

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

Example 3

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

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

Discussion

- The big-step operational semantics is not well suited for tracking step-by-step how evaluation by MiniOcaml proceeds.
- It is quite convenient, though, for proving that the evaluation of a function for particular argument values terminates:

  For that, it suffices to prove that there are values to which the corresponding function calls can be evaluated ...

Discussion (cont.)

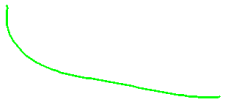
- The big-step semantis also allows to verify that optimizing transformations are correct, i.e., preserve the semantics.
- Finally, it can be used to prove the correctness of assertions about functional programs !
- The big-step operational semantics suggests to consider expressions as specifications of values.
- Expressions which evaluate to the same values, should be interchangeable ...

Caveat

- In MiniOcaml, equalitiy between values can only be tested if these do not contain functions !!
- Such values are called comparable. They are of the form

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

$e_1 = e_2 \Rightarrow$ true



$e_1 = e_2 \Rightarrow$ true