Script generated by TTT

Title: Info2 (15.01.2016)

Date: Fri Jan 15 08:35:04 CET 2016

Duration: 89:29 min

Pages: 29

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

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

Terminierung

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

Strukturierte Werte

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

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

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

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:

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

$$\frac{e_1 = e_2 \Rightarrow \texttt{true}}{e_1 = e_2 \qquad e_i \text{ terminieren}}$$

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:

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

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

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

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

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

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

Vereinfachung lokaler Definitionen

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

Vereinfachung von Funktionsaufrufen

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

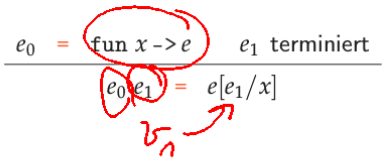
Zuerst verschaffen wir uns ein Repertoir von Umformungsregeln, die die Gleichheit von Ausdrücken auf Gleichheiten anderer, möglicherweise einfacherer Ausdrücke zurück führt ...

Vereinfachung lokaler Definitionen

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

Vereinfachung von Funktionsaufrufen

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



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:

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

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

$$\frac{e_0 \;=\; \texttt{fun } x_1 \ldots x_k \texttt{->} e \qquad e_1, \ldots, e_k \text{ terminieren}}{e_0\, e_1 \ldots e_k \;=\; e[e_1/x_1, \ldots, e_k/x_k]}$$

Diese abgeleitete Regel macht Beweise etwas weniger umständlich.

Regel für Pattern Matching

$$\frac{e_0 \;=\; []}{\texttt{match } e_0 \texttt{ with } [] \texttt{->} e_1 \mid \ldots \mid p_m \texttt{->} e_m \;=\; e_1}$$

$$\frac{e_0 \text{ terminiert} \qquad e_0 \;=\; e_1' :: e_2'}{\texttt{match } e_0 \texttt{ with } [] \texttt{->} e_1 \mid x :: xs \texttt{->} e_2 \;=\; e_2[e_1'/x, e_2'/xs]}$$

7.3 Beweise für MiniOcaml-Programme

Beispiel 1

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

Wir wollen nachweisen:

(1) `app x [] = x` für alle Listen `x`.

(2) `app x (app y z) = app (app x y) z`
für alle Listen `x, y, z`.

7.3 Beweise für MiniOcaml-Programme

Beispiel 1

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

Wir wollen nachweisen:

(1) `app x [] = x` für alle Listen `x`.

(2) `app x (app y z) = app (app x y) z` für alle Listen `x, y, z`.

Idee: Induktion nach der Länge n von `x`

$n = 0$: Dann gilt: `x = []`

Wir schließen:

```
app x []  =  app [] []
          =  match [] with [] -> [] | h::t -> h :: app t []
          =  []
          =  x
```


7.3 Beweise für MiniOcaml-Programme

Beispiel 1

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

Wir wollen nachweisen:

(1) `app x [] = x` für alle Listen `x`.

(2) `app x (app y z) = app (app x y) z` für alle Listen `x, y, z`.

Idee: Induktion nach der Länge n von `x`

$n = 0$: Dann gilt: `x = []`

Wir schließen:

```
app x []  =  app [] []
          =  match [] with [] -> [] | h::t -> h :: app t []
          =  []
          =  x
```

7.3 Beweise für MiniOcaml-Programme

Beispiel 1

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

Wir wollen nachweisen:

(1) `app x []` = `x` für alle Listen `x`.

(2) `app x (app y z) = app (app x y) z` für alle Listen `x, y, z`.

$n > 0$: Dann gilt: `x = h::t` wobei `t` Länge $n-1$ hat.

Wir schließen:

```
app x []  =  app (h::t) []
          =  match h::t with [] -> [] | h::t -> h :: app t []
          =  h :: app t []
          =  h :: t      nach Induktionsannahme
          =  x
```


Analog gehen wir für die Aussage (2) vor ...

$n = 0$: Dann gilt: `x = []`

Wir schließen:

```
app x (app y z)  =  app [] (app y z)
                 =  match [] with [] -> app y z | h::t -> ...
                 =  app y z
                 =  app (match [] with [] -> y | ...) z
                 =  app (app [] y) z
                 =  app (app x y) z
```


$n > 0$: Dann gilt: `x = h::t` wobei `t` Länge $n-1$ hat.

Wir schließen:

```
app x (app y z)  =  app (h::t) (app y z)
                 =  match h::t with [] -> [] | h::t -> h ::
                          app t (app y z)
                 =  h :: app t (app y z)
                 =  h :: app (app t y) z   nach Induktionsannahme
                 =  app (h :: app t y) z
                 =  app (match h::t with [] -> []
                       | h::t -> h :: app t y) z
                 =  app (app (h::t) y) z
                 =  app (app x y) z
```

Diskussion

- Zur Korrektheit unserer Induktionsbeweise benötigen wir, dass die vorkommenden Funktionsaufrufe terminieren.
- Im Beispiel reicht es zu zeigen, dass für alle `x, y` ein v existiert mit:

  $$\texttt{app x y} \Rightarrow v$$

  ... das haben wir aber bereits bewiesen, natürlich ebenfalls mit Induktion.

Beispiel 2

```
let rec rev = fun x -> match x
        with [] -> []
        | x::xs -> app (rev xs) [x]
let rec rev1 = fun x -> fun y -> match x
        with [] -> y
        | x::xs -> rev1 xs (x::y)
```

Behauptung

`app (rev x) y = rev1 x y` für alle Listen `x`.

Allgemeiner

`app (rev x) y = rev1 x y` für alle Listen `x, y`.

Beweis: Induktion nach der Länge n von `x`

$n = 0$: Dann gilt: `x = []`. Wir schließen:

```
app (rev x) y  =  app (rev []) y
               =  app (match [] with [] -> [] | ...) y
               =  app [] y
               =  y
               =  match [] with [] -> y | ...
               =  rev1 [] y
               =  rev1 x y
```


$n > 0$: Dann gilt: `x = h::t` wobei `t` Länge $n-1$ hat.

Wir schließen (unter Weglassung einfacher Zwischenschritte):

```
app (rev x) y  =  app (rev (h::t)) y
               =  app (app (rev t) [h]) y
               =  app (rev t) (app [h] y)    wegen Beispiel 1
               =  app (rev t) (h::y)
               =  rev1 t (h::y)              nach Induktionsvoraussetzung
               =  rev1 (h::t) y
               =  rev1 x y
```

Diskussion

- Wieder haben wir implizit die Terminierung der Funktionsaufrufe von app, `rev` und `rev1` angenommen.
- Deren Terminierung können wir jedoch leicht mittels Induktion nach der Tiefe des ersten Arguments nachweisen.
- Die Behauptung:

  ```
  rev x = rev1 x []
  ```

  folgt aus:

  ```
  app (rev x) y = rev1 x y
  ```

  indem wir: `y = []` setzen und Aussage (1) aus Beispiel 1 benutzen.

Beispiel 3

```
let rec sorted = fun x -> match x
        with x1::x2::xs -> (match x1 <= x2
                        with true -> sorted (x2::xs)
                          | false -> false)
           | _          -> true

    and merge = fun x -> fun y -> match (x,y)
        with ([],y) -> y
        |    (x,[]) -> x
        | (x1::xs,y1::ys) -> (match x1 <= y1
                        with true -> x1 :: merge xs y
                          | false -> y1 :: merge x ys
```