Script generated by TTT

Title: Seidl: Info2 (22.12.2017)

Date: Fri Dec 22 08:33:01 CET 2017

Duration: 90:01 min

Pages: 34

```
module Sort = struct
    let single list = map (fun x->[x]) list
    let rec merge l1 l2 = match (l1,l2)
        with ([],_) -> l2
        |    (_,[]) -> l1
        | (x::xs,y::ys) -> if x<y then x :: merge xs l2
                                  else y :: merge l1 ys
    let rec merge_lists = function
            [] -> [] | [l] -> [l]
    | l1::l2::ll -> merge l1 l2 :: merge_lists ll
    let sort list = let list = single list
        in let rec doit = function
           [] -> [] | [l] -> l
        |   l -> doit (merge_lists l)
    in doit list
end
```


Die Implementierung macht auch die Hilfsfunktionen single, merge und merge_lists von außen zugreifbar:

```
# Sort.single [1;2;3];;
- : int list list = [[1]; [2]; [3]]
```

Damit die Funktionen single und merge_lists nicht mehr exportiert werden, verwenden wir die Signatur:

```
module type Sort = sig
    val merge : 'a list -> 'a list -> 'a list
    val sort : 'a list -> 'a list
  end
```

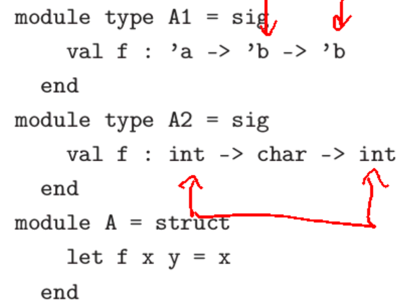

Die Funktionen single und merge_lists werden nun nicht mehr exportiert.

```
# module MySort : Sort = Sort;;
module MySort : Sort
# MySort.single;;
Unbound value MySort.single
```

Signaturen und Typen

Die in der Signatur angegebenen Typen müssen Instanzen der für die exportierten Definitionen inferierten Typen sein.

Dadurch werden deren Typen spezialisiert:



```
module type A1 = sig
    val f : 'a -> 'b -> 'b
  end
module type A2 = sig
    val f : int -> char -> int
  end
module A = struct
    let f x y = x
  end
```



```
# module A1 : A1 = A;;
Signature mismatch:
Modules do not match: sig val f : 'a -> 'b -> 'a end
                              is not included in A1
Values do not match:
  val f : 'a -> 'b -> 'a
is not included in
  val f : 'a -> 'b -> 'b
# module A2 : A2 = A;;
module A2 : A2
# A2.f;;
- : int -> char -> int = <fun>
```


6.3 Information Hiding

Aus Gründen der Modularität möchte man oft verhindern, dass die Struktur exportierter Typen einer Struktur von außen sichtbar ist.

Beispiel

```
module ListQueue = struct
  type 'a queue = 'a list
  let empty_queue () = []
  let is_empty = function
      [] -> true | _  -> false
  let enqueue xs y = xs @ [y]
  let dequeue (x::xs) = (x,xs)
end
```


Mit einer Signatur kann man die Implementierung einer Queue verstecken:

```
module type Queue = sig
  type 'a queue
  val empty_queue : unit -> 'a queue
  val is_empty : 'a queue -> bool
  val enqueue : 'a queue -> 'a -> 'a queue
  val dequeue : 'a queue -> 'a * 'a queue
end
```

Mit einer Signatur kann man die Implementierung einer Queue verstecken:

```
module type Queue = sig
  type 'a queue
  val empty_queue : unit -> 'a queue
  val is_empty : 'a queue -> bool
  val enqueue : 'a queue -> 'a -> 'a queue
  val dequeue : 'a queue -> 'a * 'a queue
end
```



```
# module Queue : Queue = ListQueue;;
module Queue : Queue
# open Queue;;
# is_empty [];;
This expression has type 'a list but is here used with type
  'b queue = 'b Queue.queue
```

Das Einschränken per Signatur genügt, um die wahre Natur des Typs queue zu verschleiern.

Soll der Datentyp mit seinen Konstruktoren dagegen exportiert werden, wiederholen wir seine Definition in der Signatur:

```
module type Queue =
sig
   type 'a queue = Queue of ('a list * 'a list)
   val empty_queue : unit -> 'a queue
   val is_empty : 'a queue -> bool
   val enqueue : 'a -> 'a queue -> 'a queue
   val dequeue : 'a queue -> 'a option * 'a queue
end
```


module $M = F(X_1)(X_2)(X_3)$

6.4 Funktoren

Da in Ocaml fast alles höherer Ordnung ist, wundert es nicht, dass es auch Strukturen höherer Ordung gibt: die Funktoren.

- Ein Funktor bekommt als Parameter eine Folge von Strukturen;
- der Rumpf eines Funktors ist eine Struktur, in der die Argumente des Funktors verwendet werden können;
- das Ergebnis ist eine neue Struktur, die abhängig von den Parametern definiert ist.

Wir legen zunächst per Signatur die Eingabe und Ausgabe des Funktors fest:

```
module type Decons = sig
  type 'a t
  val decons : 'a t -> ('a * 'a t) option
end
module type GenFold = functor (X:Decons) -> sig
  val fold_left : ('b -> 'a -> 'b) -> 'b -> 'a X.t -> 'b
  val fold_right : ('a -> 'b -> 'b) -> 'a X.t -> 'b -> 'b
  val size : 'a X.t -> int
  val list_of : 'a X.t -> 'a list
  val iter : ('a -> unit) -> 'a X.t -> unit
end
...
```



```
...
module Fold : GenFold = functor (X:Decons) ->
struct
let rec fold_left f b t = match X.decons t
    with None -> b
    | Some (x,t) -> fold_left f (f b x) t
let rec fold_right f t b = match X.decons t
    with None -> b
    | Some (x,t) -> f x (fold_right f t b)
let size t = fold_left (fun a x -> a+1) 0 t
let list_of t = fold_right (fun x xs -> x::xs) t []
let iter f t = fold_left (fun () x -> f x) () t
end;;
```

Jetzt können wir den Funktor auf eine Struktur anwenden und erhalten eine neue Struktur ...


```
module MyQueue = struct open Queue
  type 'a t = 'a queue
  let decons = function
     Queue([],xs) -> (match rev xs
          with [] -> None
          | x::xs -> Some (x, Queue(xs,[])))
   | Queue(x::xs,t) -> Some (x, Queue(xs,t))
end

module MyAVL = struct open AVL
  type 'a t = 'a avl
  let decons avl = match extract_min avl
      with (None,avl) -> None
       | Some (a,avl) -> Some (a,avl)
end
```



```
module FoldAVL = Fold (MyAVL)
module FoldQueue = Fold (MyQueue)
```

Damit können wir z.B. definieren:

```
let sort list = FoldAVL.list_of (
                    AVL.from_list list)
```

Achtung

Ein Modul erfüllt eine Signatur, wenn er sie implementiert !

Es ist nicht nötig, das explizit zu deklarieren !!

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:

`Test.cmo`	Bytecode für die Struktur
`Test.cmi`	Bytecode für das Interface
`a.out`	lauffähiges Programm

ocamlopt

- 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 ...

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;

Dieses Fragment von Ocaml nennen wir MiniOcaml.

Ausdrücke in MiniOcaml lassen sich durch die folgende Grammatik beschreiben:

$$
\begin{aligned}
E \;::=\;& \mathsf{const} \mid \mathsf{name} \mid \mathsf{op}_1\, E \mid E_1\, \mathsf{op}_2\, E_2 \mid \\
& (E_1,\ldots,E_k) \mid \texttt{let}\ \mathsf{name} = E_1\ \texttt{in}\ E_0 \mid \\
& \texttt{match}\ E\ \texttt{with}\ P_1\ \texttt{->}\ E_1\ \texttt{|}\ \ldots\ \texttt{|}\ P_k\ \texttt{->}\ E_k \mid \\
& \texttt{fun}\ \mathsf{name}\ \texttt{->}\ E \mid E\ E_1
\end{aligned}
$$

$$
P \;::=\; \mathsf{const} \mid \mathsf{name} \mid (P_1,\ldots,P_k) \mid P_1 \,\texttt{::}\, P_2
$$

Dieses Fragment von Ocaml nennen wir MiniOcaml.

Ausdrücke in MiniOcaml lassen sich durch die folgende Grammatik beschreiben:

$$
\begin{aligned}
E \;::=\;& \mathsf{const} \mid \mathsf{name} \mid \mathsf{op}_1\, E \mid E_1\, \mathsf{op}_2\, E_2 \mid \\
& (E_1,\ldots,E_k) \mid \texttt{let}\ \mathsf{name} = E_1\ \texttt{in}\ E_0 \mid \\
& \texttt{match}\ E\ \texttt{with}\ P_1\ \texttt{->}\ E_1\ \texttt{|}\ \ldots\ \texttt{|}\ P_k\ \texttt{->}\ E_k \mid \\
& \texttt{fun}\ \mathsf{name}\ \texttt{->}\ E \mid E\ E_1
\end{aligned}
$$

$$
P \;::=\; \mathsf{const} \mid \mathsf{name} \mid (P_1,\ldots,P_k) \mid P_1 \,\texttt{::}\, P_2
$$

Abkürzung

$$
\texttt{fun}\ x_1\ \texttt{->}\ \ldots \texttt{fun}\ x_k\ \texttt{->}\ e \quad\equiv\quad \texttt{fun}\ x_1\ \ldots\ x_k\ \texttt{->}\ e
$$

Dieses Fragment von Ocaml nennen wir MiniOcaml.

Ausdrücke in MiniOcaml lassen sich durch die folgende Grammatik beschreiben:

$$
\begin{aligned}
E \;::=\;& \mathsf{const} \mid \mathsf{name} \mid \mathsf{op}_1\, E \mid E_1\, \mathsf{op}_2\, E_2 \mid \\
& (E_1,\ldots,E_k) \mid \texttt{let}\ \mathsf{name} = E_1\ \texttt{in}\ E_0 \mid \\
& \texttt{match}\ E\ \texttt{with}\ P_1\ \texttt{->}\ E_1\ \texttt{|}\ \ldots\ \texttt{|}\ P_k\ \texttt{->}\ E_k \mid \\
& \texttt{fun}\ \mathsf{name}\ \texttt{->}\ E \mid E\ E_1
\end{aligned}
$$

$$
P \;::=\; \mathsf{const} \mid \mathsf{name} \mid (P_1,\ldots,P_k) \mid P_1 \,\texttt{::}\, P_2
$$

Abkürzung

$$
\texttt{fun}\ x_1\ \texttt{->}\ \ldots \texttt{fun}\ x_k\ \texttt{->}\ e \quad\equiv\quad \texttt{fun}\ x_1\ \ldots\ x_k\ \texttt{->}\ e
$$

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?)

Dieses Fragment von Ocaml nennen wir MiniOcaml.

Ausdrücke in MiniOcaml lassen sich durch die folgende Grammatik beschreiben:

$$
\begin{array}{lcl}
E & ::= & \text{const} \mid \text{name} \mid \text{op}_1\, E \mid E_1\, \text{op}_2\, E_2 \mid \\
 & & (E_1, \ldots, E_k) \mid \texttt{let}\ \text{name} = E_1\ \texttt{in}\ E_0 \mid \\
 & & \texttt{match}\ E\ \texttt{with}\ P_1\ \texttt{->}\ E_1\ \texttt{|}\ \ldots\ \texttt{|}\ P_k\ \texttt{->}\ E_k \mid \\
 & & \texttt{fun}\ \text{name}\ \texttt{->}\ E \mid E\, E_1
\end{array}
$$

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

Abkürzung

$$
\texttt{fun}\ x_1\ \texttt{->}\ \ldots \texttt{fun}\ x_k\ \texttt{->}\ e \quad \equiv \quad \texttt{fun}\ x_1\ \ldots\ x_k\ \texttt{->}\ e
$$

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 $f_1, \ldots, f_m$:

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


fun x → 1+1

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:

$$
\begin{array}{lcl}
V & ::= & \text{const} \mid \texttt{fun}\ \text{name}_1\ \ldots\ \text{name}_k\ \texttt{->}\ E \mid \\
 & & (V_1, \ldots, V_k) \mid [] \mid V_1 :: V_2
\end{array}
$$

1+1

(fun x -> x+1) 5

⇒ 6

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
```


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
```


map (fun x -> x) [1; 2; 3]

⇒ [1; 2; 3]

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 $\Longrightarrow$ 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.

Idee

- Wir definieren eine Relation: $e \Rightarrow v$ zwischen Ausdrücken und ihren Werten $\Longrightarrow$ 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.

318

Tupel

Voraus

Folgung

$$\frac{e_1 \Rightarrow v_1 \quad \ldots \quad e_k \Rightarrow v_k}{(e_1, \ldots, e_k) \Rightarrow (v_1, \ldots, v_k)}$$

Listen

$$\frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2}{e_1 :: e_2 \Rightarrow v_1 :: v_2}$$

Globale Definitionen

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