Beispiel: Endliche Automaten

Wir wollen ein Paket für endliche Automaten mit folgender Schnittstelle schreiben:

$\langle FM \rangle \equiv$

```
signature FM =
  sig
    type ''a machine          (* ''a = alphabet *)

    val make : {final :int list,
                trans :(int * ''a * int) list,
                start :int list} -> ''a machine
    val show : ''a machine
               -> {final :int list,
                   trans :(int * ''a * int) list,
                   start :int list}
    val states : ''a machine -> int

    val Atom : ''a -> ''a machine
    val Plus : ''a machine list -> ''a machine
    val Times : ''a machine list -> ''a machine
    val Star : ''a machine -> ''a machine

    val accept : ''a machine -> ''a list -> bool

    val determine : ''a machine -> ''a machine
    (* val minimize : ''a machine -> ''a machine *)
  end;
```

Der Typ der Maschine ist für Benutzer nicht erkennbar.

Implementierung der Automaten

$\langle Fm \rangle \equiv$

```
structure Fm :> FM =
  struct
    datatype 'a machine =
      Ma of {states : int,
             start : int Set.t,
             trans : (int * 'a * int) Set.t,
             final : int Set.t}

    fun make {trans=I, start=S, final=F} =
      let
        val maxStart = foldr Int.max 0 S
        val maxFinal = foldr Int.max maxStart F
        val maxState =
          foldr (fn ((i,a,j),k) =>
                 Int.max (Int.max(i,j),k)) maxFinal I
      in
        Ma{states = maxState+1,
           start = Set.mk S,
           trans = Set.mk I,
           final = Set.mk F}
      end

    fun show M = {start = start M,
                  trans = trans M,
                  final = final M}
    and start  (Ma{start = S, ...}) = Set.list S
    and final  (Ma{final = F, ...}) = Set.list F
    and trans  (Ma{trans = I, ...}) = Set.list I
    and states (Ma{states = n, ...}) = n
```

Atomare Automaten

$\langle Fm \rangle{+}{\equiv}$

```
fun Atom a = Ma{states = 2,
                start = Set.mk[0],
                trans = Set.mk[(0,a,1)],
                final = Set.mk[1]}
```

Parallelschaltung von Automaten

$\langle Fm \rangle{+}{\equiv}$

```
fun Plus [] = Ma{states = 0,
                 start = Set.mk[],
                 trans = Set.mk[],
                 final = Set.mk[]}
  | Plus (M :: Ms) = plus (M, Plus Ms)

and plus (Ma{states = nA, start = sA,
             trans = trA, final = finA},
          Ma{states = nB, start = sB,
             trans = trB, final = finB}) =
  Ma{states = nA+nB,
     start = Set.union
             sA (Set.map (addNat nA) sB),
     trans = Set.union
             trA (Set.map (incr nA) trB),
     final = Set.union
             finA (Set.map (addNat nA) finB)}

and incr (n:int) = (fn (x,a,y) => (x+n,a,y+n))
and addNat (n:int) = (fn m => m+n)
```

Hintereinanderschaltung von Automaten

$\langle Fm \rangle{+}\equiv$

```
fun Times [] = Ma{states = 1,
                  start = Set.mk[0],
                  trans = Set.mk[],
                  final = Set.mk[0]}
  | Times [M] = M
  | Times (M :: Ms) = times (M, Times Ms)

and times (Ma{states = nA, start = sA,
              trans = trA, final = finA},
           Ma{states = nB, start = sB,
              trans = trB, final = finB}) =
  let
    val sB' = Set.map (addNat nA) sB
    val trB' = Set.map (incr nA) trB
    val finB' = Set.map (addNat nA) finB
    fun between ((p,a,q)::I) NewTrans =
        if Set.member q finA
        then Set.union (Set.map (fn s => (p,a,s)) sB')
                       (between I NewTrans)
        else between I NewTrans
      | between [] NewTrans = NewTrans
    val trAB = between (Set.list trA) Set.empty
  in
    Ma{states = nA+nB,
       start = if Set.intersects sA finA
                 then Set.union sA sB' else sA,
       trans = Set.Union [trA,trAB,trB'],
       final = finB'}
  end
```

Rückkopplung eines Automaten

$\langle Fm \rangle{+}{\equiv}$

```
fun Star M =
  let
    val sA' = map (addNat 1) (start M)
    val trA' = map (incr 1) (trans M)
    val finA' = map (addNat 1) (final M)

    fun after ((p,a,q)::I) NewTrans =
      if member q finA'
        then after I
          (map (fn s => (p,a,s)) sA') @ NewTrans
      else after I NewTrans
      | after [] NewTrans = NewTrans
  in
     Ma{states = 1 + (states M),
        start = Set.mk(0 :: sA'),
        trans = Set.mk(trA' @ (after trA' [])),
        final = Set.mk(0 :: finA')}
  end
```

Deterministisch und minimal machen

$\langle Fm\ Aufgabe \rangle{\equiv}$

```
fun determine M = ..
and minimize M = ...
```

Akzeptieren eines Eingabeworts

$\langle Fm \rangle {+}{\equiv}$

```
local
  fun member x l = List.exists (fn y => y = x) l
  and add q [] = [q]
    | add q (u::U) = if q=u then (u::U)
                     else (u::(add q U))
  and union [] U = U
    | union (q::Q) U = union Q (add q U)
  and intersects [] Q = false
    | intersects (p::P) Q =
     if member p Q then true
     else intersects P Q
  and Union [] = []
    | Union (Q::Qs) = union Q (Union Qs)
in
  fun nextstates M q a =
    let fun g [] acc = acc
          | g ((x,b,p)::r) acc =
      if (x,b) = (q,a)
        then g r (add p acc) else g r acc
    in g (trans M) []
    end
  and reach M Q [] = Q
    | reach M Q (a::l) =
      let val Qs = map (fn q => (nextstates M q a)) Q
      in reach M (Union Qs) l end
  and accept M w =
      intersects (reach M (start M) w) (final M)
end
```
end

Deterministischer Automat

$\langle Fm \rangle{+}{\equiv}$

```
fun labels M = labls (trans M) []
and labls ((x,a,y)::Trans) acc =
    labls Trans (add a acc)
  | labls [] acc = acc

and power M =
  let
    fun newStateTrans A =
        foldr (action A) ([],[]) (labels M)
    and action A (a,(States,Transs)) =
        let val B = reach M A [a]
        in (add B States, add (A,a,B) Transs) end

    and service ([],Done,Trans) = ([],Done,Trans)
      | service (A::Queue,Done,Trans) =
        let val (nStates,nTrans) = newStateTrans A
            val Queue' =
                union (difference nStates Done) Queue
            and Done'  = add A Done
            and Trans' = union nTrans Trans
        in service (Queue',Done',Trans') end

    val (_,pStates,pTrans) = service ([start M],[],[])
    val pFinal =
        List.filter (intersects (final M)) pStates
  in
    {start = [start M], trans = pTrans,
     final = pFinal, states= pStates}
  end
```

```
  and determineWithStateAssoc M =
    let
      val {start = pStart, trans = pTrans,
           final = pFinal, states= pStates} =
          power M

      fun enum X = num X 0
      and num [] n = []
        | num (x::X) n = (n,x)::(num X (n+1))

      val stateAssoc = (enum pStates)

      fun nr S = renum stateAssoc S
      and renum [] y = raise Undefined
        | renum ((n,x)::L) y =
          if y=x then n else renum L y

    in
      (make {start = map nr pStart,
             trans = map (fn (p,a,q)
                          => (nr p, a, nr q)) pTrans,
             final = map nr pFinal},
       stateAssoc)
    end
  and determine M =
      (#1 (determineWithStateAssoc M))

end
```