Kapitel 8: ML-Modulsystem

Andreas Abel

LFE Theoretische Informatik
Institut für Informatik
Ludwig-Maximilians-Universität München

20. Juni 2011

Einführende Fragen

- In der Dokumentation der Standardbibliothek steht:

```
signature LIST
structure List :> LIST
```

  Was bedeutet das??
- Wie unterteile ich meine Software in sinnvolle Teilstücke?
- Was sind **Schnittstellen**?
- Was ist **Datenabstraktion** und was geht sie mich an?

Inhalt

Aufgaben von Modulsystemen

- Größere Software benötigt Struktur (kein Spaghetti-code!).
- Sinnvolle Untergliederung in Teilprojekte (**Module**), die **separat kompiliert** werden können.
- Module organisieren den **Namensraum**.
- In der *SML Basis Library* (Standardbibliothek) wird der Bezeichner `toString` verwendet als

```
Int.toString
Char.toString
OS.Path.toString
```

- Das sind Beispiele für **qualifizierte Bezeichner** (engl. *qualified identifiers*).
- Die Standardbibliothek verfügt über ein Modul `Int`, das einen Bezeichner `toString` definiert.
- Ebenso ein Modul `OS` mit Untermodul `Path`, das eine Funktion `toString` implementiert.

Import

- Wir können den Inhalt eines Moduls mittel **open** **importieren**.

```
open OS.FileSys
```

```
opening OS.FileSys
  type dirstream = ?.POSIX_FileSys.dirstream
  val openDir : string -> dirstream
  val readDir : dirstream -> string option
  val chDir : string -> unit
  ...
  val compare : file_id * file_id -> order
```

- Damit werden alle Deklarationen des importierten Moduls in den aktuellen Namensraum kopiert.
- Bestehende Definitionen gleichen Namens werden überschrieben.
- Hinweis: es gibt *kein* "`close`" das Definitionen wieder entfernt.

Strukturen

- In SML heißen Module **Strukturen** (engl. *structures*).
- Von **open** sollte man spärlichen Gebrauch machen.
- Qualifizierte Bezeichner wie `OS.FileSys.compare` sind aussagekräftiger als unqualifizierte wie `compare`:
- Auch weiß man sofort, in welchem Modul die Definition zu finden ist.
- Lange Modulpfade kann man durch **Strukturaliase** abkürzen.

```
structure FS = OS.FileSys
val _ = FS.chDir "../examples"
```

- Eine Strukturdefinition **structure** `S = SExp` definiert einen neuen Strukturbezeichner `S` durch den Ausdruck `SExp`.
- Ein Strukturausdruck ist z.B. ein bereits vorhandener Strukturbezeichner.
- Wir können aber auch eigene Strukturen definieren.

Strukturdefinitionen

- Alle Funktionen rund um Assoziationslisten packen wir in eine neue Struktur `AssocList`.

```
structure AssocList = struct

  (* the empty association list *)
  val empty = []

  (* updating the association list *)
  fun insert (k,v) l = (k,v) :: l

  (* lookup up a key in an association list *)
  fun lookup k [] = NONE
    | lookup k ((k',v) :: l) =
        if k = k' then SOME v else lookup k l

end (* structure AssocList *)
```

Strukturdefinitionen

- Beispielanwendung von `AssocList`: Telefonbuch.

```
type telBook        = (string, int) list
val  al0 : telBook  = AssocList.empty
val  al1 : telBook  = AssocList.insert ("Abel, A.", 9315) al0
val  q1             = AssocList.lookup "Hofmann, M." al1
```

- **struct** ... **end** darf beliebige Deklarationen einschliessen, auch wieder Strukturen.

```
structure S = struct                (* anything goes      *)
  open OS.Process                   (* an import          *)
  structure AL = AssocList          (* a structure alias  *)
  local  val bla = ...              (* a local block      *)
  in     fun blurb = ...
  end
end
```

Strukturtypen: Signaturen

- Die Struktur `AssocList` bekommt von SML diesen Typ:

```
structure AssocList :
  sig
    val empty  : 'a list
    val insert : 'a * 'b -> ('a * 'b) list -> ('a * 'b) list
    val lookup : ''a -> (''a * 'b) list -> 'b option
  end
```

- Mit **`sig ... end`** wird eine **Signatur** definiert.
- Eine Signatur enthält nur die *Typen* der Deklarationen.
- Signaturen nennt man auch **Schnittstellen** (engl. *interfaces*).
- Die Schnittstelle ist der "Vertrag" zwischen Kode-Produzent (oder *provider*) und Kode-Benutzer (*user*).

Schnittstellen

- Zu einer Schnittstelle gehört eigentlich noch eine **Spezifikation** des Verhaltens, z.B. folgende Gesetze:

```
lookup k empty = NONE
lookup k (insert (k',v) l) =
  if k=k' then SOME v else lookup k l
insert (k,v) (insert (k,v') l) = insert (k,v) l
```

- SML bietet keine Möglichkeit, zu spezifizieren und zu beweisen.
- Geht bisland nur in experimentellen Sprachen wie **Agda**.
- Spezifikationen werden meist informal oder semi-formal abgefasst (z.B. in natürlicher Sprache).
- Obige Gesetze können *unit tests* unterzogen werden.

```
fun law1 k          = lookup k empty = NONE
fun law2 k k' v l   = lookup k (insert ...
val t1 = List.all law1 ["Abel, A.", "Hofmann, M."]
```

Datenabstraktion

- Die Schnittstelle von `AssocList` hat einen Schönheitsfehler.
- Sie verrät die interne Darstellung von Assoziationslisten.
- Somit kann Kode-Produzent nicht auf einen effizientere Repräsentation wie z.B. *hash maps* umstellen.
- Der Benutzerkode würde nicht mehr kompilieren ("Vertragsverletzung").
- **Sehr wichtiges** Design-Prinzip **Datenabstraktion** (engl. *data abstraction*): Verstecke die interne Repräsentation der Daten!

Abstrakte Signatur

- Signaturdefinition mit **signature** ALL_CAPS_ID = ...

```
signature ASSOCLIST = sig

  (* type of association lists *)
  type (''k, 'v) T

  (* the empty association list *)
  val empty : (''k, 'v) T

  (* updating the association list *)
  val insert : ''k * 'v -> (''k, 'v) T -> (''k, 'v) T

  (* lookup up a k in an association list *)
  val lookup : ''k -> (''k, 'v) T -> 'v option
end
```

Strukturdeklaration mit Signatur

```
structure AssocList : ASSOCLIST = struct

  (* type of association lists *)
  type (''k, 'v) T = (''k * 'v) list

  (* the empty association list *)
  val empty = []

  (* updating the association list *)
  fun insert (k,v) l = (k,v) :: l

  (* lookup up a key in an association list *)
  fun lookup k [] = NONE
    | lookup k ((k',v) :: l) =
        if k = k' then SOME v else lookup k l

end (* structure AssocList *)
```

Struktur und Datenabstraktion

- Nach Deklaration **structure** `AssocList : ASSOCLIST = ...` druckt SML den abstrakten Typen anstatt `list`.

```
AssocList.empty
```

```
val it = [] : (''a,'b) AssocList.T
```

- Allerdings wird die Typrepräsentation noch nicht völlig versteckt:

```
val bla : (int,int) AssocList.T = []
```

`(int,int) AssocList.T` wird nach außen noch mit `(int * int) list` gleichgesetzt.

- **structure** `AssocList : ASSOCLIST` ist **durchsichtig** (engl. *transparent*) in bezug auf Typdefinitionen.
- **structure** `AssocList :> ASSOCLIST` ist **"blickdicht"** (engl. *opaque*).

```
val bla : (int,int) AssocList.T = []
```

```
Error: pattern and expression in val dec don't agree [tycon mismatch]
  pattern:    (int,int) AssocList.T
  expression: 'Z list
```

Beispiel: Geordnete Typen

```
datatype CMP = LT | EQ | GT     (* result of comparison function *)

signature ORD =                 (* an ordered type is ...        *)
sig
  type T                        (* ... a type with               *)
  val  compare : T * T -> CMP   (* ... an order                  *)
end

structure IntOrd : ORD =        (* instance: integers            *)
struct
  type T = int
  fun  compare (x, y) =
    if x < y then LT else
    if x = y then EQ else
    GT
end
```

Beispiel: Geordnete Typen

- Weitere Instanz: Zeichen (engl. *characters*).

```
structure CharOrd : ORD =
struct
  type T = char
  fun  compare (x, y : char) =
    if x < y then LT else
    if x = y then EQ else GT
end
```

- Lexikographischer Vergleich:

```
fun compareLex cmpA cmpB ((a1,b1), (a2,b2)) =
  case cmpA (a1,a2)
    of LT => LT
     | GT => GT
     | EQ => cmpB (b1, b2)
```

- Wie nun Typ von Paaren als Instanz von `ORD` definieren?

Parametrisierte Strukturen (Funktoren)

- Wir können **Strukturen parametrisieren**, z.B. über Strukturen:

```
functor LexOrd (structure A : ORD; structure B : ORD) : ORD =
struct
  type T = A.T * B.T

  fun compare (p1 : T, p2 : T) =
    compareLex A.compare B.compare (p1, p2)
end
```

- SML verwendet das Schlüsselwort **functor** für Strukturbezeichner mit Parametern.

	keine Par.	Parameter
Wert	**val**	**fun**
Struktur	**structure**	**functor**

- **Übung:** Implementieren Sie die Struktur der lex. geordneten Listen.

Exkurs: Funktoren

- **Funktor** ist ein Begriff aus der **Kategorientheorie**.
- Eine *Kategorie* ist eine Sammlung von ähnlichen Strukturen.
  1. Kategorie der Gruppen.
  2. Kategorie der Körper.
  3. Kategorie der Monoide.
  4. Kategorie der geordneten Mengen (`ORD`).
- Eine Kategorie ist vergleichbar einer Signatur.
- Ein Funktor ist eine Abbildung von einer Kategorie in eine andere.
  1. *Vergesslicher* Funktor von der Kategorie der Gruppen in die der Monoide: Macht aus einer Gruppe ein Monoid durch "Vergessen" der Invertierungsoperation.
  2. Spiegelfunktor von der Kategorie der geordneten Mengen in diesselbe: Macht *größer-gleich* aus *kleiner-gleich*.

Endliche Abbildungen über geordneten Typen

- Wir generalisieren Assoziationslisten zu **endlichen Abbildungen**.

```
signature MAP = sig
  type key
  type value
  type T

  val empty  : T
  val insert : key * value -> T -> T
  val lookup : key -> T -> value option
end
```

- Mögliche Implementierungen von *maps*:
  1. Assoziationslisten
  2. Suchbäume und balancierte Suchbäume (*tree maps*)
  3. Hashtabellen (*hash maps*)

Assoziationslisten, Redux

```
functor ListMap (structure K : ORD; type v) : MAP = struct

  type key                    = K.T
  type value                  = v
  type T                      = (key * value) list

  val   empty                 = []

  fun   insert (k,v) l        = (k,v) :: l

  fun   lookup k []           = NONE
    |   lookup k ((k',v)::l)  =
          case K.compare (k,k')
            of EQ  => SOME v
             | _   => lookup k l
end
```

Suchbäume

```
functor TreeMap (structure K : ORD; type v) : MAP = struct
 type key                                        = K.T
 type value                                      = v
 datatype T                                      = Empty
                                                 | Node of T * key * value * T


 val empty                                       = Empty

 fun leaf (k, v)                                 = Node (Empty, k, v, Empty)

 fun lookup k Empty                              = NONE
   | lookup k (Node(l,k',v,r))                   =
       case K.compare (k,k')
         of EQ => SOME v
          | LT => lookup k l
          | GT => lookup k r
```

Suchbäume

```
  fun insert (k,v) Empty                    = leaf (k,v)
    | insert (k,v) (Node(l,k',v',r))  =
        case K.compare (k,k')
          of EQ => Node (l, k, v, r)
           | LT => Node (insert (k,v) l, k', v', r)
           | GT => Node (l, k', v', insert (k,v) r)

end
```

- `insert` erzeugt einen geordneten Baum.
- Ist der neue Schlüßel kleiner als der der Wurzel, wird in den linken Teilbaum eingefügt. Analog: größer $\Longrightarrow$ rechts.

Funktorinstanziierung

- Durch Instanziierung eines Funktors erhalten wir eine Struktur.
- Die Parameter werden durch Zuweisung übergeben.

```
structure M = TreeMap (structure K = IntOrd; type v = string)

val m = foldl
  (fn (k,v) => M.insert k v)
  M.empty
  [(3,"A"),(4,"B"),(1,"C"),(2,"D")]

val c = M.lookup 1 m

structure M : MAP
val m = Node (Node (Empty,1,"C",Node #),3,"A",Node (Empty,4,"B"
val c = SOME "C" : M.value option
```

Zusammenfassung

- `Quali.Fied.identifier` Bezeichner mit Strukturpfad.
- **`open`** `Structure.Path` Import.
- **`signature`** `SIG =` **`sig`** `...` **`end`** Signaturdefinition.
- **`structure`** `Struct =` **`struct`** `...` **`end`** Strukturdefinition.
- **`structure`** `Struct : SIG = ...` transparente Definition.
- **`structure`** `Struct :> SIG = ...` opake Definition.
- **`functor`** `Fun (<params>) =` **`struct`** `...` **`end`** parametrisierte Struktur (genannt Funktor).
- **`structure`** `S = Fun (`**`structure`** `A = ...;` **`type`** `t = ...)` Funktorinstanziierung.