5.2 Module in SML

Algorithmen (Funktionsdeklarationen) arbeiten über Datenelementen, die zu Trägermengen zusammengefaßt werden können. Für die Formulierung von Algorithmen sind neben den speziellen Datenelementen die verfügbaren Funktionen entscheidend.
Man faßt die auftretenden Mengen und Funktionen zu

Strukturen

zusammen; die durch eine Struktur definierten Symbole nennt man *Signatur*. Man erhält dadurch ein Modulkonzept, in dem ein *Modul* durch eine Struktur und die *Schnittstelle* des Moduls durch die Signatur dargestellt wird.

Ziele der Modularisierung sind
- Kapselung von Information
  um Typen und Funktionen nur in gewünschter Weise verwenden zu können und schwer entdeckbare Fehler zu vermeiden
- Wiederverwendung von Code
  durch Packen von Daten und Algorithmen in Module mit wohldefinierten Schnittstellen

5.2.1 Signaturen

Die Signatur einer Datenstruktur gibt die Schnittstelle eines Moduls oder Subsystems, d.h. die verfügbaren Typen und Funktionen an. Häufig ist es für den Programmierer nicht wichtig und nicht wünschenswert, die genaue Implementierung der Signatur zu kennen.
Z.B.
(1) wenn die Signatur eine Maschinenstruktur beschreibt, deren genaue Implementierung maschinenabhängig ist. Programme, die Implementierungseigenschaften ausnutzen, sind nur schwer portierbar.
(2) wenn ein Programm Implementierungseigenschaften nicht ausnutzt, lassen sich ineffiziente Implementierungen (einfacher) durch effiziente ersetzen.
(3) Modularisierung der Programmierung.
(4) Schutz gegenüber dem Kunden.

Definition: *Signatur*
Eine Signatur $\Sigma = (S,F)$ besteht aus:
- einer Menge S von Typsymbolen (bzw. Sorten; Symbole für Mengen) und
- einer Menge F von Funktionssymbolen (Symbole für Werte).

Jedes $f \in F$ hat einen Typausdruck als Typ, der aus
* (Produkt),
$\rightarrow$,
Typvariablen und
Elementen aus S
aufgebaut ist.

Beispiel: Signatur für eine zahlartige Struktur: NUM.

S_{NUM} = {num},

F_{NUM} = {add, mult},

wobei add: num * num → num und

mult: num * num → num.

Beispiel: Keller

Signatur:

Typ:

stack

Konstante:

EmptyStack: stack

Funktionssymbole:

Push : int × stack → stack

pop : stack → stack

top : stack → int

isempty : stack → bool

In SML wird eine Signatur

Σ = <S, F> mit

S = s_1, …, s_n und

F = {f_1: τ_1, …, f_m: τ_m}

folgendermaßen dargestellt:

```
signature Σ =
 sig
  type s1
  …
  type sn
  val f1: τ1
  …
  val fm: τm
 end;
```

In SML werden vordefinierte Typen und Konstanten automatisch zu jeder Signatur gezählt; deshalb können bool, int, 'a list etc. immer verwendet werden, im Gegensatz zu allgemein definierten Signaturen.

Beispiel: (1) Signatur Σ_{NUM} der zahlartigen Strukturen:

```
signature NUMSIG =
    sig
        type num
        val add: num * num -> num
```

```
            val mult: num * num -> num
        end;
```

(2) Signatur der Kellerstruktur

```
signature STACKSig =
    sig
        type stack
        exception Pop and Top
        val EmptyStack: stack
        val Push: int * stack -> stack
        val pop: stack -> stack
        val top: stack -> int
        val isempty: stack -> bool
    end;
```

5.2.2 Strukturen

Eine Struktur ist eine Kollektion von Typen, Funktionen, Ausnahmen und anderen Elementen, die eingekapselt werden sollen. Die Definitionen dieser Elemente erscheinen in der Struktur.
Eine Strukturdefinition in SML hat die Form

structure <Name> =
struct
<Deklarationen>
end

Die Elemente einer Struktur können optional durch Semikolons getrennt werden.
Unter den Deklarationen können natürlich

Typdeklarationen	(der Form	`type s = τ`),
Wertdeklarationen	(der Form	`val c = e`),
Funktionsdeklarationen	(der Form	`fun f x = e`),
Ausnahmedeklarationen	(der Form	`exception E`) und selbst wieder
Strukturdeklarationen	(der Form	`structure D1 = e`)

auftreten. Vordefinierte Typen und Funktionen können ohne explizite Deklaration verwendet werden. Eine Typdeklaration

```
type s = int * int
```

drückt aus, daß s und int * int die gleichen Mengen bezeichnen. Man erhält z.B. für

```
val x:s = (3, 7); val y:s = (3, 7); x = y;
```

den Wert `true`.

(1) Zahlartige Strukturen:

```
signature NUMSIG =
 sig
          type num
```

```
            val add: num * num -> num
            val mult: num * num -> num
    end;
(a) structure NUM1: NUMSIG =
    struct
            type num = int
            fun add(x, y) = x + y
            fun mult(x, y) = x * y
    end;
```

(b) Die Struktur N2 kann in SML folgendermaßen definiert werden.

```
    structure NUM2: NUMSIG =
    struct
            type num = int * int
            fun add ((z1, n1) (z2, n2)) =
                 (z1 * n2 + z2 * n1, n1 * n2)
            fun mult ((z1, n1) (z2, n2)) = (z1 * z2, n1 * n2)
    end;
```

Um auf eine Funktion f oder einen Typ s aus einer Struktur D (außerhalb von D) zugreifen zu können, schreibt man D.f bzw. D.s.
Dadurch werden Namenskonflikte zwischen gleichbezeichneten Symbolen aus verschiedenen Strukturen vermieden.
Mithilfe des Befehls

```
    open D
```

öffnet man eine Struktur D und kann dann auf f bzw. s direkt zugreifen. Allerdings muß man hierbei auf mögliche Namenskonflikte achten.

Beispiel: Keller

(1)
```
structure STACK1 =
    struct
    datatype stack = EmptyStack |
                     Push of int * stack;
    exception popError and topError;
    fun pop EmptySack = raise popError
      | pop(Push(n, s)) = s;
   end;
```

Qualifizierter Zugriff

```
    STACK1.EmptyStack;
```

Direkter Zugriff auf die Symbole von STACK1 mit open:

```
    open STACK1;
    EmptyStack;
```

Eine andere Realisierung von STACKsig mittels Listen ist:

```
structure STACK:STACKSig =
   struct
    datatype stack = Stack of int list
    exception Pop and Top
    val EmptyStack = Stack nil
    fun Push(n, Stack s) = Stack (n::s)
    fun pop(Stack nil) = raise Pop
     | pop(Stack(n::s)) = Stack s
    fun top(Stack nil) = raise Top
     | top(Stack(n::s)) = n
    fun isempty(Stack nil) = true
     | isempty(Stack(n::s)) = false
   end;
```

Hier kommt die Hilfsfunktion `stack` vor, die nicht in `STACKsig` auftritt.
Um genau die richtige Schnittstelle zu erhalten, kann man die Struktur durch Angabe der gewünschten Signatur auf die richtige Schnittstelle einschränken.

```
structure STACK2 : STACKSig = STACK
```

Bemerkung: Signatureinschränkungen dienen zum
- Verstecken von Symbolen
- Spezialisieren von Funktionalitäten

Beispiel:

```
structure A: SIG =
    struct
   fun f (x) = nil;
    end
```

wobei

```
signature SIG =
    sig
      f: int list -> int list
    end
```

spezialisiert den Typ

`´a -> ´b list` von f

auf `int list -> int list`

Bemerkung: Abstrakte Datentypen
Es gibt in SML ein Konstrukt, das die Implementierung des Datentyps und der (im <Datentyp-Ausdruck> deklarierten) Funktionen automatisch nach außen versteckt:

```
abstype <Typname> = <Datentyp-Ausdruck>
```

```
        with
            <Deklarationen>
        end;
```

Von außen kann weder auf die Konstruktoren des <Datentyp-Ausdruck> noch auf die Werte des abstrakten Typs <Typname> direkt zugegriffen werden. Das Programm kann nur die deklarierten Symbole verwenden.

Beispiel: Keller

```
    abstype stack = Stack of int list
        with
            exception popError and topError
            EmptyStack = Stack nil
            ...
        end;
```

Hier wird der Konstruktor `Stack` automatisch versteckt und ist deshalb von außen nicht zugreifbar.

5.2.3 Charakteristische Eigenschaften

Die Angabe der Signatur genügt im allgemeinen nicht, um die gewünschten Eigenschaften einer Struktur festzulegen. Deshalb gibt man zusätzlich Beschreibungen der charakteristischen Eigenschaften an. Solche Beschreibungen werden heute meist in natürlicher Sprache abgefaßt; besser ist es, formale axiomatische Spezifikationen der charakteristischen Eigenschaften anzugeben.
Wir schreiben diese in der Prädikatenlogik erster Stufe, meist als Gleichungen.
Wir definieren die Gültigkeit von Eigenschaften in einer Struktur folgendermaßen:
Definition:
Sei A eine Formel mit einer Variablen x vom Typ T. Sei NF_T die Menge der Normalformen von T gemäß einer Struktur S.
Die Formel A **ist gültig** in S (S erfüllt A), wenn für alle $n \in NF_T$ die Formel A[n/x] in S gilt, d.h. wenn die Gültigkeit von A[n/x] aus den Deklarationen der Funktionen von S folgt.
Beispiel: Charakteristische Eigenschaften von Kellern sind

$\forall$ n: int, s: stack
(a) top(Push(n, s)) = n
(b) pop(Push(n, s)) = s

Der Nachweis der charakteristischen Eigenschaften für die Struktur

(1) STACK1
folgt trivial aus den Deklarationen von top und pop.

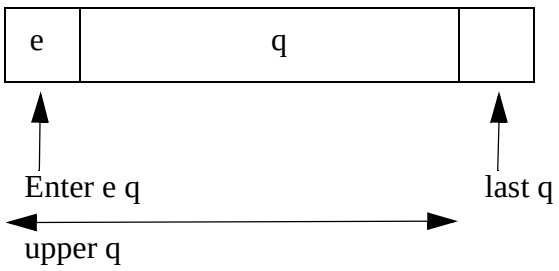
(2) STACK
Jede Normalform vom Typ stack in STACK1 hat die Form Stack l mit $l \in NF_{int\ list}$.

(a) Sei $s \in NF_{stack}$, $n \in NF_{int}$, mit $s \equiv$ Stack l
top(Push(n, s)) = [Def Push]
top(Stack(n: :l)) = [Def top]
n

(b) Sei $s \equiv$ Stack $l \in NF_{stack}$, $n \in NF_{int}$
pop(Push(n, s)) = [Def Push]
pop(Stack(n: :l)) = [Def pop]
Stack l = [Def s]
s

Definition: (Spezifikation, Realisierung)
(1) Eine (formale) **Spezifikation** SP besteht aus der Angabe einer Signatur und der Angabe von charakteristischen Eigenschaften der Signatur.
(2) Eine Struktur S **realisiert** (oder **implementiert**) eine Spezifikation SP, wenn
(i) die Signatur von S eine Erweiterung der Signatur von SP ist.
(ii) S jedes Axiom von SP erfüllt.
Zum Beispiel realisieren die Strukturen STACK1 und STACK die Spezifikation von Kellern bestehend aus der Signatur STACKSig und den oben genannten charakteristischen Eigenschaften.

Beispiel: Schlangen
Schlangen sind lineare Strukturen, bei denen Elemente “links” eingefügt und “rechts” weggenommen werden.



Die folgende Signatur und Axiome sind eine Spezifikation für Schlangen:

```
- signature QUEUESig =
   sig
     type ’a queue
     exception Last
```

```
    exception Upper
    val  EmptyQueue: 'a queue
    val  Enter: 'a * 'a queue -> 'a queue
    val  last: 'a queue -> 'a
    val  upper: 'a queue -> 'a queue
    val  isempty: 'a queue -> bool
  (* axioms:
    last  EmptyQueue = Last
    last (Enter (x, EmptyQueue)) = x
    last (Enter (x, Enter(y, q)) = last (Enter (y, q))
    upper (EmptyQueue) = Upper
    upper (Enter (x, EmptyQueue)) = EmptyQueue
    upper (Enter (x, Enter (y, q))) =
                  Enter(x, upper (Enter(y, q)))
    isempty EmptyQueue = true
    isempty (Enter (x, EmptyQueue)) = false *)
    end;
```

Die folgende Struktur `QUEUE` realisiert die Spezifikation von Schlangen:

```
- structure QUEUE:QUEUESig =
    struct
    datatype 'a queue = Queue of 'a list
    exception Last and Upper
    val EmptyQueue = Queue nil
    fun Enter(x, Queue xs) = Queue(x::xs)
    fun last (Queue nil) = raise Last
         | last (Queue (x :: nil)) = x
         | last (Queue (x :: y :: q)) = last (Queue (y :: q))
    fun upper (Queue nil) = raise Upper
         | upper(Queue (x :: nil)) = Queue nil
         | upper(Queue (x :: y :: q)) =
               Enter (x, upper(Queue(y :: q)))
    fun isempty (Queue nil) = true
     |  isempty (Queue (x :: q)) = false
   end;
```

Eine effizientere Implementierung erhält man durch eine Repräsentation einer Schlange durch ein Paar von Listen (entry, exit).

Elemente werden in entry eingefügt und aus exit entfernt. Ist exit leer, wird entry revertiert und dann werden entry und exit vertauscht:

```
(* Hilfsfunktion zum revertieren *)
- fun rev nil = nil
    | rev (x :: l) = rev l @ [x];
```

```
- structure QUEUE2:QUEUESig =
    struct
    datatype 'a queue = Queue of ('a list * 'a list)
    val EmptyQueue = Queue (nil, nil)
    fun Enter (x, (Queue (el, xl))) = Queue (x::el, xl)
    exception Last and Upper
    fun last(Queue(nil, nil)) = raise Last
      | last(Queue(e::el, nil)) = last(Queue(nil,rev(e::el)))
      | last(Queue(_, x::_)) = x
    fun upper(Queue(nil, nil)) = raise Upper
      | upper(Queue(e::el,nil))= upper(Queue(nil,rev(e::el)))
      | upper(Queue(el, x::xl))= Queue(el, xl)
    fun isempty(Queue(nil, nil)) = true
      | isempty(Queue(_, _)) = false
    end;
```

Zur Korrektheit von QUEUE gegenüber der Spezifikation:
zeigen wir nur die Axiome für last:
(1) last(Enter(x, EmptyQueue)) = x
Beweis: Sei T ein beliebiger Typ.

Für alle $x \in NF_T$

last(Enter(x, EmptyQueue)) = [Def EmptyQueue]
last(Enter(x, Queue nil)) = [Def Enter]
last(Queue(x: : nil)) = [Def last]
x

(2) last(Enter(x, Enter(y, q)) = last(Enter(y, q))
Beweis: Sei T ein beliebiger Typ.

Für alle $x, y \in NF_T$, $q \equiv$ Queue $l \in NF_{T\ queue}$, wobei $l \in NF_{T\ list}$:

last(Enter(x, Enter(y, q))) = [Def q]
last(Enter(x, Enter(y, Queue l))) = [Def Enter]
last(Enter(x, Queue(y: : l))) = [Def Enter]
last(Queue(x: :y: : l))) = [Def last]
last(Queue(y: : l))) = [Def Enter]
last(Enter(y, Queue l)) = [Def Queue l]
last(Enter(y, q))

Bemerkung:
Der oben gewählte Korrektheitsbegriff für Strukturen ist sehr streng. Im Allgemeinen fordert man für einen Typ T nicht, daß eine Formel A[u/x] für alle Normalformen u: T der realisierenden Struktur gelten muß, sondern nur daß A[u/x] für alle diejenigen Normalformen gilt, die Werte des abstrakten Typs repräsentieren. Außerdem kann die Erfüllung der

Gleichheit zur einer Äquivalenzrelation abgeschwächt werden.
Ein Beispiel ist die Realisierung von endlichen Mengen natürlicher Zahlen durch Listen, wobei eine Liste $l = [x_1, ..., x_n]$ nur dann eine Menge repräsentiert, wenn sie schwach geordnet ist, d.h. wenn

$$x_1 \leq x_2 \leq ... \leq x_n$$

gilt.
Z.B. repräsentiert die Liste [1, 2, 3] die Menge {1, 2, 3}.
Aber nicht alle Listen repräsentieren Mengen. Z. B. ist die Liste [2, 1, 3] kein Repräsentant, da sie nicht schwach geordnet ist.
Außerdem hat jede Menge unendlich viele Repräsentanten, die alle bzgl. der Mengenoperationen äquivalent sind.
Die Menge {1, 2, 3} wird z. B. durch die Listen

[1, 2, 3], [1, 1, 2, 3], [1, 2, 2, 3, 3, 3] etc.

repräsentiert.

Zusammenfassung

- **Signaturen** beschreiben Schnittstellen.
  Axiome beschränken die möglichen Implementierungen auf die angegebenen charakteristischen Eigenschaften.
  Eine **formale Spezifikation** besteht aus der Angabe einer Signatur und Axiomen.

- **Strukturen** sind Module und können auf geeignete Schnittstellen eingeschränkt werden.

- **Abstrakte Datentypen** verstecken die Werte eines Datentyps.

- Die **Korrektheit** einer Struktur oder eines abstrakten Datentyps kann in vielen Fällen durch Gleichungsbeweise gezeigt werden.