Die Modulsprache von *SML*

Größere Programme baut man *modular* auf, d.h. gliedert sie in einzelne, sinnvoll abgeschlossene Programm-*Module*.

Ein Modul hat eine *Schnittstelle*, in der festgelegt wird, welche Teile des Moduls von außen verfügbar sind.

Terminologie: In SML heißen Module *Strukturen* und Schnittstellen *Signaturen*.

Zwischen Kernsprache und Modulsprache besteht grob folgende Analogie:

	Objekte	Beschreibungen	Berechungen
Kernsprache	Werte	Typen	Funktionen
Modulsprache	Strukturen	Signaturen	Funktoren

Zur Compilezeit wird

1. die Signatur von Strukturen berechnet und
2. überprüft, ob bei Funktoranwendungen die Signaturen der Argumentstrukturen zu den Argumentsignaturen der Funktoren passen.

Zur Laufzeit werden durch Anwendung von Funktoren neue Module berechnet und zu größeren Programmen verbunden.

Syntax der Modulsprache

Bezeichnungen:

x	Wertvariable,	*con*	Wertkonstante,	t	(Wert-)Terme,
α	Typvariable	*tyc*	Typkonstruktoren	τ	Typen
Σ	Signatur-,	X	Struktur-,	*Fid*	Funktorvariable

Signaturausdrücke

SigDef	:=	`signature` $\Sigma = Sig$	(Definition von Signaturen)
Sig	:=		(Signaturen:
		Σ	Signaturvariable
	$\mid$	`sig` *Dec* … *Dec* `end`	einfache Signatur
	$\mid$	*Sig* `where`	
		`type` $(\alpha, \ldots, \alpha)$ $Tyc = \tau$	Typeinschränkung)
Dec	:=		(Deklaration von:
		`val` $x : \tau$	Wertvariablen
	$\mid$	`type` $(\alpha, \ldots, \alpha)$ *tyc*	Typkonstruktoren
	$\mid$	`datatype` $(\alpha, \ldots, \alpha)$ *tyc* =	Datentypkonstruktoren
		con `of` $\tau \mid \ldots \mid$ *con* `of` τ	und Wertkonstanten
	$\mid$	`structure` $X : Sig$	Strukturvariablen
	$\mid$	`include` *Sig*	Signaturerweiterungen
	$\mid$	*Dec* `sharing type` $Tyc = Tyc$	Typkonstruktorgleichheiten)
A	:=	$X \mid A.X$	(rel. Strukturvariable)
Tyc	:=	*tyc* $\mid$ $A.Tyc$	(rel. Typkonstruktoren)

Strukturausdrücke

			(Strukturen:
Str	$:=$	A	Strukturvariable
	$\mid$	$\texttt{struct}\ \mathit{Def}\ \dots\ \mathit{Def}\ \texttt{end}$	einfache Struktur
	$\mid$	$\mathit{Fid}(\mathit{Str})$	Funktoranwendung
	$\mid$	$\texttt{let}\ \ \texttt{structure}\ X = \mathit{Str}$ $\texttt{in}\ \mathit{Str}\ \texttt{end}$	lokale Strukturdefinition
	$\mid$	$\mathit{Str} : \mathit{Sig}$	transparente Einschränkung
	$\mid$	$\mathit{Str} :> \mathit{Sig}$	opake Einschränkung)

Def	$:=$		(Definition von:
		$\texttt{val}\ x : \tau = t$	Werten
	$\mid$	$\texttt{type}\ (\alpha, \dots, \alpha)\ \mathit{tyc} = \tau$	Typkonstruktoren
	$\mid$	$\texttt{datatype}\ (\alpha, \dots, \alpha)\ \mathit{tyc}\ =$ $\mathit{con}\ \texttt{of}\ \tau \mid \dots \mid \mathit{con}\ \texttt{of}\ \tau$	Datentypen
	$\mid$	$\texttt{structure}\ X = \mathit{Str}$	Strukturen)

Funktordefinitionen

$\mathit{FuncDef}$	$:=$	$\texttt{functor}\ \mathit{Fid}(X : \mathit{Sig}) = \mathit{Str}$	(Def. von Funktoren)
Fid	$:=$	$F_0 \mid F_1 \mid \dots$	(Funktorvariable)

Beispiele für Strukturen

Multimengen

Hier ist eine Schnittstelle und ein Modul für Multimengen:

$\langle BAG \rangle \equiv$

```
signature BAG =
  sig
    type 'a t   (* Multimengen von 'a-Objekten *)
    val empty : 'a t
    val add : ''a * ''a t -> ''a t
    val count : ''a * ''a t -> int
  end;

structure Bag =
  struct
     type 'a t = ('a * int) list
     val empty = []
     fun add (a,[]) = [(a,1)]
       | add (a,((b,n)::rest)) =
         if a=b then ((b,n+1)::rest)
         else (b,n)::(add (a,rest))
     fun count (a,[]) = 0
       | count (a,((b,n)::rest)) =
         if a=b then n else count (a,rest)
     fun union [] bs = bs
       | union ((a,n)::rest) ((b,m)::bs) =
         if a=b then union rest ((a,n+m)::bs)
         else union rest ((b,m)::(union [(a,n)] bs))
       | union ((a,n)::rest) [] = ((a,n)::rest)
  end
```

Allgemeinste Signatur einer Struktur

Bei der Auswertung der Strukturdeklaration

```
structure Bag = struct ... end
```

wird automatisch eine (allgemeinste) Signatur berechnet, nämlich

$\langle\rangle\equiv$
```
structure Bag :
  sig
    type 'a t = ('a * int) list
    val add : ''a * (''a * int) list
              -> (''a * int) list
    val count : ''a * (''a * int) list -> int
    val empty : 'a list
    val union : (''a * int) list -> (''a * int) list
                -> (''a * int) list
  end
```

Diese automatisch berechnete Signatur enthält mehr Information als die oben definierte Signature `BAG`:

- sie enthält die Typabkürzung,
- sie benutzt die Typabkürzung in den Typen nicht,
- sie enthält alle Komponenten der Struktur, auch `union`.

Transparente Struktureinschränkung

Man kann bei der Definition einer Struktur eine Signatur vorgeben, wie in

⟨⟩+≡
```
structure Bag' : BAG = struct ... end
structure Bag' = struct ... end : BAG
```

Man kann auch eine schon definierte Struktur wie `Bag` auf die in `BAG` angegebene Schnittstelleninformation beschränken:

⟨⟩+≡
```
structure Bag' = Bag : BAG;
```

Dadurch wird eine wie folgt eingeschränkte Struktur erzeugt:

- die Struktur enthält nur die in der Signatur erwähnten Komponenten:

  ⟨⟩+≡
  ```
  - Bag'.union;
  Error: unbound variable or constructor:
         union in path Bag'.union
  ```

- die Typen der Werte sind wie in der Signatur angegeben,

  ⟨⟩+≡
  ```
  - Bag'.add;
  val it = fn : ''a * ''a Bag.t -> ''a Bag.t
  ```

- *Transparenz*: Typabkürzungen bleiben sichtbar: es bleibt erkennbar, daß `'a Bag.t = ('a * int) list`:

  ⟨⟩+≡
  ```
  val mybag = [(true,3),(false,2)] : bool Bag'.t;
  ```

Opake Struktureinschränkung

Will man die Implementierungstypen von *Str* verbergen, so benutzt man die restriktivere Einschränkung *Str* `:>` *Sig*,

- entweder bei der Definition

  ⟨⟩+≡
  ```
  structure Bag'' :> BAG = struct ... end
  structure Bag'' = struct ... end :> BAG
  ```

- oder zur Einschränkung einer schon definierten Struktur:

  ⟨⟩+≡
  ```
  structure Bag'' :> BAG = Bag;
  structure Bag'' = Bag :> BAG;
  ```

Jetzt wird die Typabkürzung aus `Bag` verborgen, und damit auch der Aufbau der Objekte durch Wertkonstruktoren:

⟨⟩+≡
```
- Bag''.empty;
val it = - : 'a Bag''.t
- Bag'.empty;
val it = [] : 'a Bag.t
```

Man kann also die Listenkonstruktoren nicht mehr benutzen, um einen `'a Bag''.t` zu bilden:

⟨⟩+≡
```
- val mybag = [(true,3),(false,2)] : bool Bag''.t;
Error: expression doesn't match constraint [tycon mismatch]
  expression: (bool * int) list
  constraint: bool Bag''.t
```

Simulation abstrakter Datentypen durch opake Struktureinschränkung

Abstakte Datentypen kann man durch opake Struktureinschränkungen *Str* `:>` *Sig* simulieren: die Deklaration

⟨⟩+≡
```
abstype ''a bag = Bag of ...
   with
        fun f = ...  : ... ''a bag ...
    end
```

entspricht der Struktur

⟨⟩+≡
```
structure A :> sig type ''a bag
                   val f : ...
               end =
   struct
       datatype ''a bag = Bag of ...
       fun f = ... : ... ''a bag ...
   end
```

Auf die in der Struktur `A` als Komponente enthaltene Funktion `f` kann man mit dem *langen (relativierten, qualifizierten) Namen* `A.f` zugreifen.

Die Struktur `A` kann man durch

⟨*open*⟩≡
```
- open A;
```

öffnen; danach kann man den *kurzen Namen* `f` statt `A.f` benutzen.

Abstrakte Datentypen: `abstype`

Abstrakte Datentypen verbergen wie opake Struktureinschränkungen die Wertkonstruktoren:

⟨*abstype*⟩≡
```
abstype ''a bag = Bag of (''a * int) list
   with
     val empty = Bag []
     fun add (a, Bag []) = Bag [(a,1)]
       | add (a, Bag ((b,n)::rest)) =
         if a=b then Bag((b,n+1)::rest)
         else
             let val (Bag bs) = add (a, Bag rest)
             in
                 Bag ((b,n)::bs)
             end
     fun count (a, Bag []) = 0
       | count (a, Bag ((b,n)::rest)) =
         if a=b then n else count (a,Bag rest)
     fun union (Bag []) bs = bs
       | union bs (Bag []) = bs
       | union (Bag ((a,n)::rest)) (Bag ((b,m)::bs)) =
         if a=b then union (Bag rest) (Bag ((a,n+m)::bs))
         else
             let val (Bag newbs) =
                 union (Bag [(a,n)]) (Bag bs)
             in
                 union (Bag rest) (Bag ((b,m)::newbs))
             end
   end
```

Die Auswertung der Deklaration ergibt:

⟨*abstype*⟩+≡

```
type 'a bag
val empty = - : 'a bag
val add = fn : ''a * ''a bag -> ''a bag
val count = fn : ''a * ''a bag -> int
val union = fn : ''a bag -> ''a bag -> ''a bag
```

Einen `''a bag` kann man nur mit `empty` und `add` erzeugen; der *Typkonstruktor* `Bag` ist außerhalb der Deklaration nicht verfügbar:

⟨⟩+≡

```
- Bag [];
Error: unbound variable or constructor: Bag
```