Öffnen von Strukturen

Um nicht immer den Strukturnamen verwenden zu müssen, kann man auch alle Definitionen einer Struktur auf einmal sichtbar machen:

```
- open Pairs;
opening Pairs
   type 'a Pair = 'a * 'a
   val Pair : 'a * 'b -> 'a * 'b
   val first : 'a * 'b -> 'a
   val second : 'a * 'b -> 'b
- Pair;
val it = fn : 'a * 'b -> 'a * 'b
- Pair (4,3);
val it = (4,3) : int * int
```

Öffnen von Strukturen

Strukturen öffnen sollte man aber möglichst nur lokal tun, um Namenskonflikte mit anderen offenen Modulen zu vermeiden:

- Für Ausdrücke:

```
let open <struct>
in <expr>
end
```

- Für Definitionen:

```
local open <struct>
in <defs>
end
```

Geschachtelte Strukturen

Strukturen können selbst auch wieder Strukturen enthalten:

```
structure Quads =
  struct
    structure Pairs =
      struct
        type 'a Pair = 'a * 'a
        fun Pair(a,b) = (a,b)
        fun first(a,_) = a
        fun second(_,b) = b
      end
    type 'a Quad = 'a Pairs.Pair Pairs.Pair
    fun Quad (a,b,c,d) =
        Pairs.Pair (Pairs.Pair(a,b),Pairs.Pair(c,d))
    fun first q = Pairs.first(Pairs.first q)
    fun second q = Pairs.second(Pairs.first q)
    fun fourth q = Pairs.second(Pairs.second q)
  end
```

Geschachtelte Strukturen

```
- Quads.Quad(1,2,3,4);
val it = ((1,2),(3,4)) : (int * int) * (int * int)
- Quads.Pairs.first;
val it = fn : 'a * 'b -> 'a
```

4.13.2 Signaturen

Mithilfe von Signaturen kann man einschränken, was ein Modul nach außen exportiert:

```
structure Count =
  struct
    val cnt = ref 0
    fun setCounter n  = cnt := n
    fun getCounter () = !cnt
    fun incCounter () = cnt := !cnt+1
  end

- !Count.cnt;
val it = 0 : int
```

Der Zähler ist nach außen sichtbar, zugreifbar und sogar veränderbar.

Signaturen

Will man, daß nur über die von der Struktur selbst definierten Funktionen auf ihn zugegriffen werden kann, tut man dies mit einer Signatur:

```
signature Count =
  sig
    val setCounter : int -> unit
    val incCounter : unit -> unit
    val getCounter : unit -> int
  end
```

Die Signatur enthält den Counter selbst nicht.

Signaturen

Nun kann man mit dieser Signatur einer Struktur eine eingeschränkte Schnittstelle zuweisen:

```
- structure SafeCount = Count:Count;
structure SafeCount : Count
- open SafeCount;
opening SafeCount
  val setCounter : int -> unit
  val incCounter : unit -> unit
  val getCounter : unit -> int
- SafeCount.cnt;
stdIn:1.1-1.14 Error: unbound variable or constructor:cnt in path SafeCount.cnt
```

Die Signatur bestimmt also, welche Definitionen exportiert werden.

Signaturen

Eine eingeschränkte Schnittstelle zuweisen geht auch schon direkt bei der Definition:

```
structure Count1 : Count =
  struct
    val cnt = ref 0
    fun setCounter n  = cnt := n
    fun getCounter () = !cnt
    fun incCounter () = cnt := !cnt+1
  end

- !Count1.cnt;
stdIn:196.2-196.12 Error: unbound variable or constructor: cnt in path Count1.cnt
```

Signaturen

Die in der Signatur angegebenen Typen müssen Instanzen der für die exportierten Definitionen inferierten Typen sein: Dadurch werden deren Typen spezialisiert:

```
signature A1 = sig val f : 'a -> 'b -> 'b end
signature A2 = sig val f : int -> char -> int end
structure A = struct fun f x y = x end
- structure A1 = A:A1;
stdin: Error: value type in structure doesn't match signature spec name: f
   spec: 'a -> 'b -> 'b
   actual: 'a -> 'b -> 'a
- structure A2 = A:A2;
structure A2 : A2
- A2.f;
val it = fn : int -> char -> int
```

Information Hiding

Aus Gründen der Modularität möchte man oft nicht, dass die Struktur der Typen, die ein Modul zur Verfügung stellt, nach außen bekannt ist. Beispiel:

```
structure ListQueue =
  struct
    exception EmptyQueue
    type 'a Queue = 'a list
    val empty = nil
    fun isempty nil = true
      | isempty _   = false
    fun enqueue nil y = [y]
      | enqueue (x::xs) y = x :: enqueue xs y
    fun dequeue nil = raise EmptyQueue
      | dequeue (x::xs) = (x,xs)
  end
```

Information Hiding

Will man verstecken, dass eine Queue eine Liste ist, kann man das mit einer Signatur:

```
signature Queue =
  sig
    exception EmptyQueue
    type 'a Queue
    val empty : 'a Queue
    val isempty : 'a Queue -> bool
    val enqueue : 'a Queue -> 'a -> 'a Queue
    val dequeue : 'a Queue -> 'a * 'a Queue
  end
```

Information Hiding

Das Einschränken per Signatur genügt nicht, um die wahre Natur des Typs `Queue` zu verschleiern.

```
- structure Queue = ListQueue : Queue;
structure Queue : Queue

- open Queue;
opening Queue
  exception EmptyQueue
  type 'a Queue = 'a list
  val empty : 'a Queue
  val isempty : 'a Queue -> bool
  val enqueue : 'a Queue -> 'a -> 'a Queue
  val dequeue : 'a Queue -> 'a * 'a Queue
- isempty nil;
val it = true : bool
```

Information Hiding

Um die Implementierung der Datentypen zu verstecken, muss man das sogenannte *opaque signature matching* (`:>` anstatt `:`) verwenden:

```
- structure HiddenQueue = ListQueue :> Queue;
structure HiddenQueue : Queue
```

Durch die Verwendung von `:>` werden alle exportierten Typen, deren Definition nicht explizit in der Signatur steht, abstrahiert.

Information Hiding

```
- open HiddenQueue;
opening HiddenQueue
  exception EmptyQueue
  type 'a Queue
  val empty : 'a Queue
  val isempty : 'a Queue -> bool
  val enqueue : 'a Queue -> 'a -> 'a Queue
  val dequeue : 'a Queue -> 'a * 'a Queue
```

Information Hiding

Die Struktur `HiddenQueue` ist ein abstrakter Datentyp:

```
- isempty empty;
val it = true : bool

- isempty nil;
stdIn:301.1-301.12 Error: operator and operand don't agree [tycon mismatch]
  operator domain: 'Z Queue
  operand: 'Y list
  in expression:
    isempty nil
```

4.13.3 Funktoren

Funktoren sind parametrisierte Modulen:

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

Funktoren

Man legt zunächst per Signatur die Eingabe und Ausgabe des Funktors fest:

```
signature Enum =
  sig
    type Enum
    val null : Enum
    val incr : Enum -> Enum
  end
signature Counter =
  sig
    type Counter
    val setCounter : Counter -> unit
    val incCounter : unit -> unit
    val getCounter : unit -> Counter
  end
```

Funktoren

```
functor GenCounter (structure Enum : Enum) : Counter =
  struct
    open Enum
    type Counter = Enum
    val cnt = ref null
    fun setCounter x = cnt := x
    fun incCounter() = cnt := incr(!cnt)
    fun getCounter() = !cnt
  end
```

- Die Anwendung des Funktors `GenCounter` auf eine Struktur mit der Signatur `Enum` erzeugt eine neue Struktur mit der Signatur `Counter`

Funktoren

```
structure IntEnum =
  struct
    type Enum = int
    val null = 0
    fun incr x = x+1
  end

structure IntCounter = GenCounter(structure Enum = IntEnum);
structure IntCounter : Counter

- IntCounter.setCounter 5;
val it = () : unit
- IntCounter.incCounter();
val it = () : unit
- IntCounter.getCounter ();
val it = 6 : IntCounter.Counter
```

Funktoren

- Eine erneute Anwendung des Funktors erzeugt eine neue Struktur:

```
structure StringEnum =
  struct
    type Enum = string
    val null = "a"
    fun incr str =
     let val cs = String.explode str
         val new = case cs of nil => [#"a"]
                            | #"z"::_ => #"a"::cs
                            | c::cs' => chr(ord c+1)::cs'
     in String.implode new
     end
  end
- structure StringCounter =
    GenCounter (structure Enum = StringEnum);
```

Funktoren

```
- StringCounter.incCounter();
val it = () : unit
- StringCounter.getCounter();
val it = "b": StringCounter.Counter
```

Funktoren

- Mit einem Funktor kann man sich auch mehrere Instanzen des gleichen Moduls erzeugen:

```
- structure CountApples =
    GenCounter (structure Enum = IntEnum);
structure CountApples : Counter
- structure CountPears =
    GenCounter (structure Enum = IntEnum);
structure CountPears : Counter
```

Funktoren

```
- CountApples.incCounter();
val it = () : unit
- CountApples.incCounter();
val it = () : unit
- CountPears.incCounter  ();
val it = () : unit
- CountApples.getCounter();
val it = 2 : CountApples.Counter
- CountPears.getCounter();
val it = 1 : CountPears.Counter
```

4.13.4 Sharing Constraints

Manchmal ist es notwendig, dass zwei Typen, die in verschiedenen Eingabe-Strukturen eines Funktors definiert sind, identifiziert werden:

```
signature Printable =
  sig
    type Printable
    val print : Printable -> unit
  end
signature PrCounter =
  sig
    type Counter
    val setCounter : Counter -> unit
    val incCounter : unit -> unit
    val getCounter : unit -> Counter
    val prtCounter : unit -> unit
  end
```

Sharing Constraints

- Wir wollen jetzt einen Funktor definieren, der eine Struktur produziert, die nicht nur zählen, sondern auch den Zählerstand ausdrucken kann:

```
functor GenPrCounter (structure Enum : Enum
                      structure Prt  : Printable)
  : PrCounter =
  struct
    structure Cnt = GenCounter (structure Enum = Enum)
    open Cnt
    fun prtCounter() = Prt.print (getCounter())
  end
```

Sharing Constraints

- Das System antwortet mit der folgenden Fehlermeldung:

```
stdIn:845.26-845.50 Error: operator and operand
don't agree [tycon mismatch]
  operator domain: ?.Printable
  operand:         Counter
  in expression:
    Prt.print (getCounter ())
```

- Weil `Enum` und `Prt` zwei verschiedene Strukturen sind, weiß der Compiler nicht, dass die Typen `Enum.Enum` und `Prt.Printable` gleich sein sollen. Das muss man ihm mithilfe von Sharing Constraints explizit mitteilen.

Sharing Constraints

```
functor GenPrCounter (structure Enum : Enum
                                and Prt  : Printable
        sharing type Enum.Enum = Prt.Printable)
  : PrCounter =
  struct
    structure Cnt = GenCounter (structure Enum = Enum)
    open Cnt
    fun prtCounter () = Prt.print (getCounter())
  end
```

Der Funktor darf nun allerdings nur noch auf solche Strukturen angewendet werden, bei denen die Constraints erfüllt sind.

4.14 Programmieren im Großen

Längere Programme geben wir nicht auf der interaktiven Kommando-Zeile ein, sondern legen sie aus Dateien ab. Den Inhalt einer Datei können wir z.B. mithilfe der Funktion `use : string -> unit` einlesen, übersetzen und in der SML-Umgebung sichtbar machen:

```
- use "test.sml";
[opening test.sml]
  type s = int
  type t = bool
  val x = 1 : int
  val y = 2 : int
val it = () : unit
```

Projekte in mehreren Dateien

Größere Programmier-Projekte sind jedoch i.a. auf mehrere Dateien, wenn nicht gar mehrere Verzeichnisse verteilt.

- Regeln:
  - ▷ jede Struktur bzw. jeder Funktor liegen in einer separaten Datei;
  - ▷ mehrfach verwendete Signaturen sollten in eigenen Dateien gesammelt werden;
  - ▷ zusammengehörige Projekt-Bestandteile kommen in ein gemeinsames Verzeichnis.

Projekte in mehreren Dateien

- Besteht ein Projekt aus mehr als zwei Dateien, ist es offenbar mühsam, sich die Datei-Namen zu merken und die entsprechenden Folgen von `use`-Aufrufen zu verwalten. Dazu dient die Struktur `CM` , der Compilation Manager, der bei der SML-Implementierung von New Jersey dabei ist.

Bibliotheken

```
Library
    signature Counter
    structure IntCounter
is
    counter.sig
    intcounter.sml
    foo.sml
```

Die Bibliothek stellt die Signatur `Counter` sowie die Struktur `IntCounter` bereit. Zu deren Herstellung dienen die nach dem "is" aufgelisteten Dateien (die "Members").

Zum Laden von Bibliotheken bietet CM folgende Funktionen an:

```
make : unit -> unit
make' : string -> unit
```

Bibliotheken

- Die Funktion `make` sucht eine Datei "sources.cm" und versucht, aus der dort angegebenen Spezifikation eine Bibliothek herzustellen. `make'` funktioniert genauso, benutzt stattdessen aber den angegebenen String als Datei-Namen.
- Um Bibliotheken effizient herstellen zu können, verwaltet CM Abhängigkeiten zwischen den in Dateien definierten Strukturen.
- Die Übersetzung der Dateien berücksichtigt diese Abhängigkeiten.

Insbesondere wird eine Datei nur dann neu übersetzt, wenn sie seit der letzten Kompilierung verändert wurde oder von Dateien abhängt, deren Modifizierung einen Einfluss haben könnten.

Bibliotheken

Die exportierten Elemente können auch innerhalb der Bibliothek von den Members benutzt werden.

```
Library
    signature BAR
    structure Foo
is
    bar.sig
    foo.sml
    cml.cm
```

Bibliotheken

Für den lokalen Gebrauch innerhalb von Bibliotheken kann man mehrere Dateien zu einer Gruppe zusammen fassen.

```
Library
    signature A
    structure A
is
    a.sig
    a.sml
    utils.cm
```

`utils.cm` könnte dann z.B. Funktionen aus `utils.sml` und Datenstrukturen aus `data.sml` zusammenfassen...

Bibliotheken

```
Group
is
    utils.sml
    data.sml
```

Sämtliche definierten Elemente werden exportiert. Soll der Export eingeschränkt werden, kann man eine explizite Export-Liste angeben:

```
Group
    structure Data
is
    utils.sml
    data.sml
```

Bibliotheken

Soll die Gruppe nur innerhalb der Bibliothek oder Gruppe `a.cm` verwendet werden, kann man diese explizit mitteilen:

```
Group (a.cm)
    structure Data
is
    utils.sml
    data.sml
```

Bibliotheken

Regeln:

- Explizit können nur Funktoren, Strukturen und Signaturen exportiert werden.
- Jede Datei darf nur einmal in einer ".cm"-Datei vorkommen.
- Gruppen und Bibliotheken können beliebig oft verwendet werden.

Bibliotheken

Weitere Features:

- automatischer Aufruf von Präprozessoren ("Tools"), die die Source-Datei erst noch erzeugen müssen;
- bedingter Export bzw. bedingter Einschluss von Members in Abhängigkeit z.B. von sml-Version oder Betriebssystem;
- Erzeugung von Stand-alone-Versionen.