Kapitel 9

Typabstraktion und Modularisierung

Programme können sehr groß werden. Für einen SML-Interpreter muss man mit etwa fünfzigtausend Zeilen Programm rechnen. Betriebssysteme wie Linux oder Windows bestehen aus vielen Millionen Zeilen Programm (Windows 98 angeblich aus etwa 50 Millionen). Solche komplexen Programme kann man nur beherrschen, wenn sie modular aufgebaut sind. Dabei zerlegt man ein Programm in eine nicht zu große Zahl von *Modulen*. Große Module zerlegt man wieder in Module, sodass die Komplexität auf jeder Gliederungsebene überschaubar bleibt. Man spricht von einem *modularen Design*. Ein vergleichbares Prinzip findet man bei der Gliederung eines Mathematik-Buchs in Teile, Kapitel, Abschnitte, Definitionen, Sätze und Beweise.

Ein Modul *implementiert* eine *Schnittstelle*. Die Schnittstelle eines Software-Moduls beschreibt, welche Typen und Prozeduren das Modul zur Verfügung stellt und was die Prozeduren berechnen. Das sogenannte *Abstraktionsprinzip* fordert, dass für Benutzung eines Moduls die Kenntnis seiner Schnittstelle hinreicht, also keine Kenntnis seiner Implementierung erforderlich ist.

Die Bedeutung des Abstraktionsprinzip lässt sich am Beispiel von komplexen technischen Geräten wie Autos, Handys oder Computer aufzeigen. Um diese zu benutzen, muss man nicht wissen, wie sie implementiert sind, es genügt ihre Schnittstelle zu verstehen. Ohne das Abstraktionsprinzip könnte man solche komplexen Geräte weder benutzen noch konstruieren. Ein einzelner Ingenieur wäre nämlich nicht in der Lage, alle Module eines Autos oder Computers zu verstehen. Also hat man viele Ingenieure, die jeweils nur für die Implementierung weniger Module zuständig sind.

Eine wichtige Qualität eines modularen Designs besteht darin, dass man die Implementierung einer Schnittstelle verbessern kann, ohne dass sich die durch die Schnittstelle zur Verfügung gestellte Funktionalität ändert. Beispielsweise kann

man eine anfänglich ineffiziente Implementierung einer Schnittstelle *A* schrittweise durch effizientere Implementierungen ersetzen. Die *A* benutzenden Module profitieren dann von diesen Änderungen, ohne dass sie irgendwie verändert werden müssen. Wenn man beispielsweise die Reifen eines Autos austauscht, sind keine Änderungen an den anderen Modulen des Autos (zum Beispiel dem Motor oder der Klimaanlage) erforderlich.

Große Softwaresysteme sind so komplex, dass sie immer Fehler enthalten. Das modulare Design eines Softwaresystems sorgt dafür, dass man auftretende Fehler einem oder einigen wenigen Modulen zuordnen kann. Nur so ist es möglich, Fehler mit erträglichem Aufwand zu finden und zu beheben.

Für die Formulierung von Modulen stellt Standard ML drei Konstruktionen zur Verfügung: Strukturen, Signaturen und Funktoren. In diesem Kapitel lernen wir diese grundlegenden programmiersprachlichen Konstruktionen mithilfe einfacher Beispiele kennen. Danach lernen wir am Beispiel des Programmiersystems Moscow ML, wie man Module in der Praxis mithilfe von Dateien und getrennter Übersetzung realisiert.

9.1 Strukturen

Strukturen realisieren in Standard ML dieselbe grundlegende Organisationsidee wie Dateiverzeichnisse in Betriebssystemen. Wir haben bereits einige Standardstrukturen kennengelernt, zum Beispiel `Math`, `List` und `TextIO`. Eine Struktur ist eine Kollektion von Objekten, die durch Bezeichner identifiziert werden und als *Felder der Struktur* bezeichnet werden. Die Standardstruktur `Math` hat unter anderem die folgenden Felder:

```
val pi  : real
val sin : real -> real
```

Die Felder einer Struktur können mithilfe von *zusammengesetzten Bezeichnern* zugegriffen werden:

```
Math.sin(Math.pi/2.0)
1.0 : real
```

Strukturen können mithilfe von *Strukturdeklarationen* eingeführt werden. Hier ist ein Beispiel:

```
structure S =
   struct
      val a = 4
      fun f x = x+1
   end
```

Diese Deklaration bindet den Bezeichner S an eine Struktur mit zwei Feldern:

```
val a : int
val f : int -> int
```

Die Felder von S können beispielsweise wie folgt benutzt werden:

```
S.a
4 : int

S.f(2*S.a + S.a)
13 : int
```

Konvention für Strukturbezeichner

Für Strukturen wählt man üblicherweise Bezeichner, die mit einem Großbuchstaben beginnen.

Jede Folge von Deklarationen kann mithilfe einer Strukturdeklaration lokalisiert und zusammengefasst werden:

```
structure S = struct Deklarationen end
```

Dabei werden die Deklarationen wie üblich ausgeführt. Für jeden durch die Deklaration gebunden Bezeichner bekommt die Struktur *S* ein entsprechendes Feld. Hier ist ein weiteres Beispiel:

```
structure S =
    struct
        val a = 4
        exception E
        datatype t = A | B
        structure T = struct val a = a+3 end
    end
```

Das Feld S.T ist wieder eine Struktur. Mit dem zusammengesetzten Bezeichner S.T.a kann man das Feld a der Struktur S.T zugreifen:

```
S.T.a
7 : int
```

Die Felder S.A und S.B sind Konstruktoren und können entsprechend verwendet werden:

```
fun f S.A = S.B
  | f S.B = S.A

val f : S.t -> S.t
```

Entsprechendes gilt für die Felder S.E (eine Ausnahme) und S.t (ein Konstruktortyp).

Einschränkend gilt, dass Strukturdeklaration nicht innerhalb von Let-Ausdrücken erlaubt sind. Außerdem sind Strukturen keine Werte. Sie können also nicht als Argumente oder Ergebnisse von Prozeduren verwendet werden.

Öffnen von Strukturen

Mithilfe der Deklaration

```
open S
```

kann man die den Feldern einer Struktur entsprechenden lokalen Bindungen auf der Ebene der Open-Deklaration sichtbar machen. Man spricht vom *Öffnen einer Struktur*. Für die obige Struktur `S` hat die Deklaration

```
open S
```

denselben Effekt wie die Deklarationsfolge

```
val a = S.a
exception E = S.E
datatype t = datatype S.t
structure T = S.T
```

Die Deklaration von `t` hat eine spezielle Form, die dafür sorgt, dass die Bezeichner `A` und `B` an die Konstruktoren `S.A` und `S.B` des Typs `S.t` gebunden werden.

Generativität

Die Deklarationen

```
exception E = S.E
datatype t = datatype S.t
```

haben eine spezielle Form (sogenannte *Synonym-Deklarationen*), die wir bisher nicht kenngelernt haben. Sie können nicht durch die Deklarationen

```
exception E
datatype t =  A | B
```

ersetzt werden, da diese eine neue Ausnahme beziehungsweise einen neuen Typ und neue Konstruktoren erzeugen (sogenannte *generative Deklarationen*).

9.2 Typabstraktion

Ein abstrakter Typ ist ein Typ, dessen Implementierung nicht sichtbar ist. Ein Beispiel ist der Typ `TextIO.instream` für Eingabeströme, den wir in Kapitel 8 kennengelernt haben. Obwohl die Implementierung von `TextIO.instream` verborgen ist, kann man mit den Werten dieses Typs sinnvoll programmieren, da die Standardstruktur `TextIO` entsprechende Operationen bereitstellt:

```
TextIO.openIn    : string -> TextIO.instream
TextIO.inputLine : TextIO.instream -> string
TextIO.closeIn   : TextIO.instream -> unit
```

Die Prozedur `TextIO.openIn` öffnet eine Datei und liefert einen Eingabestrom, mithife dessen die Datei gelesen werden kann. Die Prozedur `TextIO.inputLine` liefert die jeweils nächste Zeile auf einem Eingabestrom.

```
signature ISET =
    sig
        type set
        val  empty  : set
        val  insert : int * set -> set
        val  member : int * set -> bool
    end
```

Abbildung 9.1: Die Signatur `ISET`

Die Prozedur `TextIO.closeIn` invalidiert einen Eingabestrom und gibt die zugehörige Datei wieder frei.

Mithilfe von Strukturen kann man abstrakte Typen zusammen mit ihren Operationen einführen. Als Beispiel betrachten wir einen Typ, dessen Werte als endliche Mengen von ganzen Zahlen verstanden werden sollen. Dafür deklarieren wir zunächst, wie in Abbildung 9.1 gezeigt, eine *Signatur* `ISET`, die den abstrakten Typ zusammen mit seinen Operationen beschreibt. Bei der Zeile

```
type set
```

handelt es sich um eine *abstrakte Typspezifikation*, die festlegt, dass `set` ein Typ ohne Gleichheit ist. Um `set` als Typ mit Gleichheit zu spezifizieren (siehe Abschnitt 3.7), würde man

Abstrakte Typspezifikationen

```
eqtype set
```

schreiben. Die *Konstante* `empty` stellt die leere Menge zur Verfügung. Die Prozedur `insert` liefert zu einer Zahl x und einer Menge M die Menge $M \cup \{x\}$. Die Prozedur `member` testet, ob eine Zahl in einer Menge enthalten ist.

Für Signaturen wählt man üblicherweise Bezeichner, die nur aus Großbuchstaben bestehen.

Konvention für Signaturbezeichner

Als Nächstes deklarieren wir, wie in Abbildung 9.2 gezeigt, eine Struktur `ISet`, die die Signatur `ISET` implementiert. Wir haben eine möglichst einfache Implementierung gewählt, die Mengen als Listen repräsentiert, die mehrere Auftreten desselben Elements enthalten können. Der *Signaturconstraint*

```
ISet :> ISET
```

sorgt dafür, dass die Felder der Struktur `ISet` nur gemäß ihrer Spezifikation in der Signatur `ISET` benutzt werden können. Damit wird die Implementierung des Typs `ISet.set` verborgen.[1] Ausdrücke wie

```
ISet.member(5,[5,7])
```

[1] Man spricht auch von Enkapsulierung oder Information Hiding.

```
structure ISet :> ISET =
    struct
        type set = int list
        val  empty = nil
        val  insert = op::
        fun  member(x,s) = List.exists (fn y => y=x) s
    end
```

Abbildung 9.2: Die Struktur `ISet`

sind also unzulässig. Um die Menge {5, 7} zu bekommen, können wir den folgenden Ausdruck verwenden:

```
ISet.insert(5, ISet.insert(7, ISet.empty))
<set> : ISet.set
```

Erwartungsgemäß verrät auch der Interpreter nicht, wie die Werte von abstrakten Typen realisiert sind. Die Implementierung von abstrakten Typen bleibt also auch auf dieser Ebene verborgen.

Eine *Typabstraktion* besteht also aus einem abstrakten Typ, Konstanten und einigen Operationen. Mithilfe der Konstanten und Operationen können Werte des abstrakten Typs beschrieben und benutzt werden. Die *Schnittstelle* einer Typabstraktion wird mithilfe einer Signatur beschrieben. Die Typdiziplin einer Programmiersprache sorgt dafür, dass die Konstanten und Operationen einer Typabstraktion nur gemäß ihrer Signatur benutzt werden können. Die Implementierung von Typabstraktionen erfolgt mithilfe von Strukturen und Signaturconstraints.

Die *Semantik einer Typabstraktion* beschreibt die Semantik der Operationen der Typabstraktion. Sie wird meistens durch eine informelle Beschreibung festgelegt, wie wir das am Beispiel von `ISET` gesehen haben.[2] Präziser kann die Semantik einer Typabstraktion mithilfe einer *Modellimplementierung* beschrieben werden. Beispielsweise können wir die Struktur `ISet` als die Modellimplementierung der Typabstraktion `ISET` ansehen. Da eine Modellimplementierung nur dazu dient, die Semantik einer Typabstraktion zu spezifizieren, sollte sie so geschrieben sein, dass sie möglichst einfach zu verstehen ist. Es ist also belanglos, ob sie die Operationen der Typabstraktion effizient realisiert. Dagegen erwartet man von einer *Gebrauchsimplementierung*, dass sie die Operationen der Typabstraktion effizient implementiert.

Die Unterstützung von Typabstraktionen ist eine wichtige Qualität moderner Programmiersprachen. Typabstraktionen enkoppeln die Benutzung einer Funktionalität von ihrer Implementierung. Damit hat man die in der Praxis wichtige Mög-

[2] Wir benutzen den Namen der Signatur auch als Namen für die entsprechende Typabstraktion.

lichkeit, die Implementierung einer Funktionalität schrittweise durch effizientere Versionen zu ersetzen, ohne dass die benutzenden Programmteile dies sehen können. Beispielsweise kann man endliche Mengen mithilfe von Bäumen effizienter realisieren als mit Listen. Die Einführung einer Typabstraktion sorgt dafür, dass dieser drastische Repräsentationswechsel für die benutzenden Programmteile unsichtbar bleibt (bis auf Effizienz).

Typabstraktionen sind ein Beispiel für Schnittstellen, wie wir sie in der Einführung dieses Kapitels beschrieben haben.

Die Einführung von abstrakten Typen mithilfe eines Signaturconstraints ist *generativ*. Gegeben die Deklarationen der Signatur `ISET` und der Struktur `ISet`, bedeutet dies, dass die Deklaration

```
structure S :> ISET = ISet
```

einen neuen Typ `S.set` einführt. Folglich sind Ausdrücke wie

```
S.insert(2, ISet.empty)
```

unzulässig, da die Typen `ISet.set` und `S.set` verschieden sind. Man kann also beliebige viele verschiedene Version von Mengen mit Elementen aus `int` erzeugen. Das kann vorteilhaft sein, wenn die verschiedenen Versionen unabhängig voneinander benutzt werden sollen. Die Typprüfung sorgt dann dafür, das versehentliche Vertauschungen automatisch als Fehler erkannt werden.

Abstrakte Gleichheit versus Darstellungsgleichheit

Wir haben bereits darauf hingewiesen, dass die Signatur `ISET` den Typ `set` als einen Typ ohne Gleichheit spezifiziert. Folglich kann unsere Modellimplementierung `ISet` korrekterweise darauf verzichten, den Typ `set` mit der richtigen Gleichheit zu implementieren. Beispielsweise beschreiben die Ausdrücke

```
ISet.insert(2, ISet.insert(1, ISet.empty))
ISet.insert(1, ISet.insert(2, ISet.empty))
```

dieselbe Menge, evaluieren aber zu verschiedenen Werten ([2, 1] und [1, 2]). Man muss also zwischen *abstrakter Gleichheit* (hier der Gleichheit von Mengen) und *Darstellungsgleichheit* (hier der Gleichheit der Darstellungen von Mengen) unterscheiden. Wenn man Mengen durch sortierte Listen darstellt, kann man eine Modellimplementierung bekommen, deren Darstellungsgleichheit gleich der abstrakten Gleichheit ist. Damit ist jedoch für die Praxis nichts gewonnen, da die bekannten effizienten Darstellungen für Mengen stets eine von der abstrakten Gleichheit verschiedene Gleichheit liefern (da sie mehr als eine Darstellung für dieselbe Menge erlauben).

Das Gesagte bedeutet aber nicht, dass eine Typabstration für endliche Mengen keinen Gleichheitstest zur Verfügung stellen kann. Alle Probleme verschwinden, wenn wir den Gleichheitstest als explizite Operation

```
signature QUEUE =
  sig
    type 'a queue
    val empty  : 'a queue
    val insert : 'a * 'a queue -> 'a queue
    val head   : 'a queue -> 'a        (* Empty *)
    val tail   : 'a queue -> 'a queue  (* Empty *)
  end

structure Queue :> QUEUE =
  struct
    type 'a queue = 'a list
    val empty = nil
    fun insert (a,q) = q @ [a]
    val head = hd
    val tail = tl
  end
```

Abbildung 9.3: Die Struktur `Queue`

```
val eq : set * set -> bool
```

in die Signatur `ISET` aufnehmen. Die dafür erforderliche Erweiterung der Modellimplementierung `ISet` kann mithilfe der folgenden Deklarationen geschehen:

```
fun subset (s,s') = List.all (fn x => member(x,s')) s
fun eq (s,s') = subset(s,s') andalso subset(s',s)
```

9.3 Schlangen

Eine in der Programmierung oft vorkommende Typabstraktion sind sogenannte *Schlangen* (englisch „queues“). Abbildung 9.3 spezifiert diese Typabstraktion mithilfe einer Signatur `QUEUE` und einer Modellimplementierung `Queue`. Bei Schlangen handelt es sich um Listen, die nur mit den angegebenen Operationen konstruiert und zerlegt werden dürfen. Die Operation `insert` liefert zu einem Wert a und einer Schlange q die Schlange $q@[a]$, die q am hinteren Ende um das Element a erweitert. Die Operationen `head` und `tail` liefern den Kopf beziehungsweise des Rumpf einer Schlange.

Der naive Programmierer wird sich an dieser Stelle fragen, was denn der Nutzen der Typabstraktion `QUEUE` sein soll, da man doch Listen anstelle von Schlangen verwenden kann und dabei auch noch Schreibarbeit spart. Für die Verwendung der Typabstraktion `QUEUE` gibt es jedoch zwei gute Gründe.

Der erste Grund ist, dass die explizite Formulierung und Verwendung der Typab-

```
structure FQueue :> QUEUE =
  struct
    type 'a queue = 'a list * 'a list

    val empty = ([], [])

    fun insert (a, ([], _)) = ([a], [])
      | insert (a, (q , r)) = (q, a::r)

    fun head (q, r) = hd q

    fun tail ([x], r) = (rev r, [])
      | tail (q  , r) = (tl q, r)
  end
```

Abbildung 9.4: Eine effiziente Implementierung von Schlangen

straktion `QUEUE` die Lesbarkeit von großen Programmen verbessern kann. Außerdem sorgt die Typdisziplin dafür, dass auf Schlangen wirklich nur die vereinbarten Operationen angewendet werden können. Beides zahlt sich aus, sobald ein Programm geändert werden muss, was in der Praxis häufig der Fall ist und in der Regel durch Programmierer erfolgt, die das Programm ursprünglich nicht geschrieben haben.

Der zweite Grund wird auch dem naiven Programmierer sofort einleuchten. Man kann nämlich die Operation `insert` für Schlangen effizienter realisieren als dies mit Listen möglich ist. Die Modellimplementierung `Queue` realisiert die Operation `insert` für Schlangen der Länge n mit der Laufzeit $\Theta(n)$. Das bedeutet, dass für n aufeinanderfolgende Einfügeoperationen, ausgehend von der leeren Liste, die Laufzeit $\Theta(n^2)$ anfällt ($1+2+\cdots+n = \frac{n}{2}(n+1)$). Wir werden gleich sehen, dass dies auch mit der Laufzeit $\Theta(n)$ erreicht werden kann.

Abbildung 9.4 zeigt eine effiziente Implementierung der Typabstraktion `QUEUE`. Diese stellt eine Schlange durch ein Paar

$$(q, r)$$

von zwei Listen dar, wobei die Liste

$$q@rev(r)$$

die eigentliche Schlange repräsentiert. Nichtleere Schlangen werden dabei grundsätzlich durch ein Paar realisiert, dessen erste Liste nichtleer ist. Diese Eigenschaft der Darstellung bezeichnet man als eine *Darstellungsinvariante*. Die Darstellungsinvariante sorgt dafür, dass ein Paar (q, r) genau dann die leere Schlange

darstellt, wenn $q = []$ gilt. Die Operation `insert` kann jetzt mit konstanter Laufzeit realisiert werden. Um die Darstellungsinvariante einzuhalten, müssen zwei Fälle unterschieden werden. Aus demselben Grund muss die Operation `tail` ebenfalls zwei Fälle unterscheiden und von der Darstellung

$([x], r)$

zu der Darstellung

(*rev r*, [])

übergehen. Für diesen Fall benötigt sie lineare Laufzeit relativ zur Länge der Schlange.

Wir haben jetzt erreicht, dass die Operation `insert` konstante Laufzeit hat. Allerdings hat die Operation `tail` jetzt unter Umständen lineare Laufzeit. Das ist nicht weiter schlimm, da ein teurer Aufruf von `tail` Arbeit leistet, von der die nachfolgenden Aufrufe von `tail` profitieren. Wenn ein Aufruf von `tail` eine n-elementige Liste reversieren muss, ist sichergestellt, dass die nachfolgenden $n-1$ Aufrufe von `tail` nur jeweils konstante Laufzeit erfordern. Offensichtlich kann ein in die Schlange eingefügtes Element nur höchstes einmal in einer durch die Operation `tail` zu reversierenden Liste vorkommen. Wenn wir eine Folge von n Anwendungen der Prozeduren `insert`, `head` und `tail`, ausgehend von der leeren Schlange, betrachten, gilt, dass die für das Reversieren benötigte Laufzeit kleiner gleich der Anzahl der in der Folge vorkommenden Anwendungen von `insert` ist. Folge können wir die Folge mit der Laufzeit $O(n)$ ausführen. Akkumuliert betrachtet benötigt die Operation `tail` also nur konstante Laufzeit.

Die Korrektheit der Implementierung `FQueue` steht und fällt mit der Einhaltung der Darstellungsinvariante. Beispielsweise liefern alle drei Operationen für die inkonsistente Darstellung

([], [1, 2])

falsche Ergebnisse. Da die Typdisziplin sicher stellt, dass Darstellungen nur durch die Struktur `FQueue` gebaut werden, können wir sicher sein, dass die Darstellungsinvariante eingehalten wird. Ohne Typabstraktion und Typdisziplin hätten benutzende Programmteile die Möglichkeit, inkonsistente Darstellungen einzuschleusen. Die Erfahrung zeigt, dass dies, wenn die Möglichkeit dazu besteht, in der Praxis auch passiert (durch Fehler von gutwilligen Programmierern).

9.4 Funktoren

In Abschnitt 9.3 haben wir eine polymorphe Typabstraktion für Schlangen über einem *beliebigen* Typ beschrieben. Die Festlegung auf einen bestimmten Elementtyp haben wir dabei mithilfe eines abstrakten Typkonstruktors und mithilfe von polymorphen Operationen vermieden. Die Struktur `FQueue` stellt uns damit effiziente Schlangen für beliebige Elementtypen zur Verfügung.

In Abschnitt 9.2 haben wir eine Typabstraktion für endliche Mengen über `int` angegeben. Da man Mengen auch für andere Elementtypen als `int` benötigt, ist eine polymorphe Realisierung wünschenswert. Um zu testen, ob ein Wert Element einer Menge ist, benötigt man einen Gleichheitstest für den Elementtyp. Da Standard ML spezielle Typvariablen für Typen mit Gleichheit hat (siehe Abschnitt 3.7), gelingt es, eine polymorphe Signatur für Mengen zu schreiben und diese auch zu implementieren. Leider ist damit für die Praxis nichts gewonnen, da effiziente Implementierungen von Mengen mehr als nur einen Gleichheitstest für den Elementtyp benötigen.

In Kapitel 10 werden wir lernen, wie man Mengen über geordneten Elementtypen effizient implementieren kann. Für einen Elementtyp `elem` benötigt man dafür eine Vergleichsprozedur

```
compare : elem * elem -> order
```

die die Ordnung zur Verfügung stellt. Um die effiziente Implementierung für beliebige geordnete Elementtypen zur Verfügung zu stellen, benötigen wir einen sogenannten *Funktor*, der zu einem Typ `elem` und zu einer Vergleichsprozedur

```
val  compare : elem * elem -> order
```

eine Struktur liefert, die Mengen über `elem` implementiert. Abbildung 9.5 zeigt die Deklaration eines solchen Funktors. Da wir erst in Kapitel 10 lernen, wie man Mengen über geordneten Elementtypen effizient implementiert, implementiert der deklarierte Funktor Mengen naiv. Für die effiziente Implementierung muss lediglich der *Rumpf der Funktordeklaration* umgeschrieben werden (der durch `struct` und `end` geklammerte Teil). Der vor dem Rumpf stehende Teil der Funktordeklaration wird als *Kopf der Funktordeklaration* bezeichnet.

Die Typabstraktion `ISET` aus Abschnitt 9.2 können wir jetzt mithilfe einer *Anwendung des Funktors* `Set` implementieren:

```
structure ISet :> ISET =
  Set(type elem = int
      val compare = Int.compare)
```

```
functor Set
  (type elem
   val  compare : elem * elem -> order)
  :>
  sig
    type set
    val empty  : set
    val insert : elem * set -> set
    val member : elem * set -> bool
  end
  =
  struct
    type set = elem list
    val  empty = nil
    val  insert = op::
    fun  member(x,s) =
           List.exists (fn y => compare(x,y)=EQUAL) s
  end
```

Abbildung 9.5: Ein Funktor für Mengen

Den Signaturconstraint `:> ISET` können wir dabei auch weglassen, da die Funktoranwendung sowieso eine Struktur mit der Signatur `ISET` liefert.

Endliche Mengen über dem Typ `string` können wir mithilfe des Funktors `Set` wie folgt implementieren:

```
structure StringSet =
  Set(type elem = string
      val compare = String.compare)
```

9.5 Programmiersprachliche Details

Wir haben jetzt die grundlegenden Konstruktionen kennengelernt, mit denen Standard ML die Formulierung von modularen Programmen unterstützt. Bisher haben wir diese Konstruktionen aus Anwendungssicht mithilfe von Beispielen beschrieben. Jetzt wollen wir auf einige programmiersprachliche Aspekte eingehen.

Neben Werten und Typen gibt jetzt drei neue Arten von programmiersprachlichen Objekten, die als Strukturen, Signaturen und Funktoren bezeichnet werden. Für die neuen Arten gibt es passende Deklarationen, mit denen ihre Objekte an Bezeichner gebunden werden können. An welche Art von Objekt eine Deklaration einen Bezeichner bindet, kann man immer an dem die Deklaration einleitenden Schlüsselwort erkennen. Deklarationen für Strukturen, Signaturen und Funktoren bezeichnet man zusammenfassend auch als *Moduldeklarationen*.

Die Syntax von Standard ML ist so entworfen, dass man aus dem Kontext eines Bezeichnerauftretens ersehen kann, welche Art von Objekt gemeint ist. Folglich macht es keine Probleme, wenn ein Bezeichner gleichzeitig an einen Wert, einen Typ, eine Struktur, eine Signatur und einen Funktor gebunden ist. Der syntaktische Kontext eines benutzenden Bezeichnerauftretens legt immer eindeutig fest, für welche Art eine Bindung benötigt wird. Man sagt, dass die *Namensräume* für die verschiedenen Arten syntaktisch getrennt sind.

Die Typprüfung bestimmt für jeden Bezeichner, der an eine Struktur gebunden ist, eine Signatur, die die Felder der Struktur beschreibt. Für die Zulässigkeit einer Benutzung des Bezeichners (als Bezeichner für eine Struktur) ist die dem Bezeichner zugeordnete Signatur maßgeblich (also nicht die bindende Strukturdeklaration).

Eine Strukturdeklaration

```
structure S :> Sig = Rumpf
```

bindet den Bezeichner *S* statisch an die Signatur *Sig*. Außerdem wird statisch überprüft, dass die durch *Rumpf* beschriebene Struktur die Signatur *Sig* erfüllt. Dafür gelten die folgenden Regeln:

1. Die durch *Rumpf* beschriebene Struktur muss mindestens die Felder der Signatur *Sig* realisieren. Es ist zulässig, dass die Struktur mehr Felder realisiert als die Signatur spezifiziert. Zusätzliche Felder werden durch einen Signaturconstraint unsichtbar gemacht (was den Bezeichner *S* anbetrifft).

2. Die durch *Rumpf* beschriebene Struktur muss die Felder der Signatur *Sig* mindestens so allgemein realisieren, wie diese in *Sig* spezifiziert ist. Beispielsweise ist es zulässig, dass die durch *Rumpf* beschriebene Struktur ein von *Sig* durch

   ```
   val f : int -> int
   ```

   spezifiziertes Feld allgemeiner mit einem Feld

   ```
   val f : 'a -> 'a
   ```

   realisiert.

3. Signaturconstraints sind auch dann zulässig, wenn die Signatur keine abstrakten Typen spezifiziert. Soche Signaturconstraints können aus drei Gründen sinvoll sein: Vergessen von Feldern, Einschränkung der Allgemeinheit von Feldern, und Sicherstellung der Tatsache, dass die durch *Rumpf* beschriebene Struktur die Signatur *Sig* erfüllt.

Standard ML wäre einfacher und flexibler, wenn Strukturen Werte, Funktoren Prozeduren und Signaturen Typen wären. Diese Gleichschaltung logisch verwandter

Konzepte scheitert in der Praxis jedoch daran, dass die Typkonsistenz von Programmen dann nicht mehr in allen Fällen automatisch überprüft werden kann.

9.6 Komponenten

Große Programme zerlegt man in *Komponenten*, die getrennt auf Dateien geschrieben werden. Eine Komponente besteht aus einer *Signatur* und einer *Implementierung*. Die Implementierung deklariert eine Struktur, die die Signatur realisiert. Die Signatur und die Implementierung einer Komponente werden auf getrennte Dateien geschrieben. Bevor eine Komponente benutzt werden kann, müssen ihre Signatur und ihre Implementierung übersetzt werden. Die Signatur muss vor der Implementierung übersetzt werden. Nachdem eine Komponente übersetzt ist, kann sie wie eine Standardstruktur benutzt werden.

Die Sprache Standard ML macht keine Aussagen über Komponenten. Dies ist die Aufgabe des jeweiligen Programmiersystems. Hier beschreiben wir das Komponentensystem von Moscow ML.

Sei `A` der Name einer Komponente. Dann muss die Signatur von `A` auf einer Datei `A.sig` und die Implementierung von `A` auf einer Datei `A.sml` stehen. Die Signatur muss die Form

```
signature A = sig ... end
```

haben. Die Implementierung muss die Form

```
structure A :> A = struct ... end
```

haben. Es muss also derselbe Bezeichner für die Signatur und die Struktur einer Komponente verwendet werden. Da die Deklarationsrahmen eindeutig aus dem jeweiligen Dateinamen abgeleitet werden können, bietet Moscow ML dem Programmierer die Möglichkeit, die Rahmen wegzulassen und nur die mit ... bezeichneten Programmteile in die Dateien zu schreiben. Das ist bequem, macht es aber schwieriger, die entsprechenden Programmteile mit anderen Programmiersystemen für Standard ML zu verwenden.

Die Signatur einer Komponente `A` übersetzt man mit

```
mosmlc -c A.sig
```

Die Implementierung einer Komponente `A` übersetzt man mit

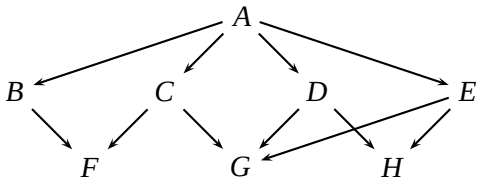
```
mosmlc -c A.sml
```

Die übersetzte Signatur einer Komponente `A` wird in eine Datei `A.ui` geschrieben, die übersetzte Implementierung in eine Datei `A.uo`.

Moscow ML bietet die Möglichkeit, die Signatur einer Komponente `A` wegzulassen. Beim Übersetzen der Implementierung von `A` wird dann automatisch eine Datei `A.ui` angelegt, die die für die Implementierung abgeleitete Signatur in übersetzter Form enthält.

Typischerweise benutzt eine Komponenten andere Komponenten. Die Abhängigkeiten zwischen Komponenten kann man durch einen *Abhängigkeitsgraphen* darstellen, bei dem die Komponenten als Knoten erscheinen. Eine Kante (A, B) bedeutet, dass die Komponente A die Komponente B benutzt. Der Abhängigkeitsgraph muss azyklisch sein. Der Abhängigkeitsgraph eines in Komponenten zerlegten Programms hat immer genau einen Knoten, von dem aus alle Komponenten des Programms erreichbar sind (im Beispielsgraph A). Dieser Knoten repräsentiert die sogenannte *Hauptkomponente*.

Abhängigkeitsgraph der Komponenten



Hauptkomponente

Die Signatur einer Komponente kann die Typen anderer Komponenten benutzen. Um eine Signatur S übersetzen zu können, müssen die Signaturen aller Komponenten, die von S benutzt werden, bereits in übersetzter Form vorliegen. Zum Übersetzen eine Signatur sind keine Implementierungen erforderlich.

Um die Implementierung I einer Komponente K übersetzen zu können, müssen die Signatur von K und die Signaturen aller Komponenten, die von I benutzt werden, in übersetzter Form vorliegen. Außer I sind keine weiteren Implementierungen erforderlich.

Komponenten werden zu einem direkt ausführbaren Programm zusammengebunden

Um ein direkt ausführbares Programm zu erzeugen, müssen die Signaturen und Implementierungen aller beteiligten Komponenten in übersetzter Form vorliegen. Ein Werkzeug, dass *Binder* (englisch „Linker“) heißt, rekonstruiert dann ausgehend von der Hauptkomponente den Abhängigkeitsgraph und ordnet die Signatur- und Strukturdeklarationen der Komponenten zu einer linearen Folge an. Die Reihenfolge der Deklarationen ist im Allgemeinen nicht eindeutig festgelegt. Es muss aber gewährleistet sein, dass Signatur- und Strukturbezeichner deklariert werden bevor sie benutzt werden. Das sorgt dafür, dass die Strukturdeklaration der Hauptkomponente ganz an das Ende der Folge kommt. Für den obigen Abhängigkeitsgraphen ist beispielsweise die folgende Reihenfolge zulässig:

F G H B C D E A

Die vorhergegangene (getrennte) Übersetzung der Deklarationen sorgt dafür, dass

die so gebildete Deklarationsfolge in jedem Fall ein zulässiges Programm ist. Aus diesem erzeugt der Binder das direkt ausführbare Programm. Man sagt, dass die übersetzten Komponenten zu einem direkt ausführbaren Programm *zusammengebunden* werden.

Wenn `H` die Hauptkomponente ist, erzeugt man das direkt ausführbare Programm zum Beispiel mit

```
mosmlc H.sml
```

Dieser Befehl bindet alle Programmteile zu einem direkt ausführbaren Programm zusammen, dass je nach Betriebssystem in die Datei `a.out` oder `mosmlout.exe` geschrieben wird.

Wenn die Implementierung einer Komponente geändert wird, genügt es, die Implementierung neu zu übersetzen und das direkt ausführbare Programm neu zu erzeugen.

Es ist auch möglich Funktoren als Komponenten zu realisieren. Dazu schreibt man die Funktordeklaration in eine Datei `X.sml` und übersetzt diese mit

```
mosmlc -toplevel X.sml
```

Architektur eines Programms

Größere Programme entwirft man auf der Ebene von Komponenten. Man überlegt sich schrittweise, welche Komponenten man haben will und welche Signaturen sie haben sollen. Den Abhängigkeitsgraphen zusammen mit den Signaturen der Komponenten bezeichnet man als die *Architektur* des Programms. Mit dem Schreiben von Implementierungen beginnt man erst, wenn die Grundzüge der Architektur feststehen. Oft gibt es einen Architekten und mehrere Programmierer, die die Komponenten schreiben. Die getrennte Übersetzung sorgt dafür, dass man Signaturen und Implementierungen sehr früh auf Wohlgetyptheit überprüfen lassen kann, eine Möglichkeit, die in der Praxis wichtig ist.

Übungen

Aufgabe 9.1 (Signaturen)

1. Betrachten Sie die Strukturdeklaration in Abbildung 9.2. Welche Signatur wird dem Bezeichner `ISet` zugeordnet, wenn man den Signaturconstaint `:> ISET` weglässt?

2. Betrachten Sie die Funktordeklaration in Abbildung 9.5. Welche Signatur ordnet die Deklaration

```
structure StringSet =
  Set(type elem = string
      val compare = String.compare)
```

dem Bezeichner `StringSet` zu?

3. Warum ist die Deklaration

```
structure S :> sig eqtype t end =
  struct type t = int -> int end
```

unzulässig?

Aufgabe 9.2 (Endliche Funktionen) In Abschnitt 9.2 haben Sie gelernt, wie man endliche Mengen implementiert. Sie sollen nun mit denselben Techniken endliche Funktionen (siehe Abschnitt 2.3) implementieren. Dies soll für einen gegebenen Typ *key* gemäß der Signatur

```
type 'a map
val empty  : 'a map
val insert : key * 'a * 'a map -> 'a map
val lookup : key * 'a map -> 'a option
```

und der folgenden Spezifikation erfolgen:

$$\begin{aligned} \mathit{empty} &= \emptyset \\ \mathit{insert}(k, a, f) &= f[a/k] \\ \mathit{lookup}(k, f) &= \text{if } k \in \mathit{Dom}\ f \text{ then } \mathit{SOME}(f(k)) \text{ else } \mathit{NONE} \end{aligned}$$

1. Deklarieren Sie eine Struktur `IMap`, die endliche Funktionen für *key* = *int* implementiert. Deklarieren Sie dazu zunächst eine passende Signatur `IMAP`.

2. Deklarieren Sie einen Bezeichner

```
val m : int IMap.map
```

der an die folgende Funktion gebunden ist:

$$\{1 \mapsto 1,\ 5 \mapsto 3,\ 6 \mapsto 0,\ 7 \mapsto 0\}$$

3. Geben Sie Typ- und Wertdeklarationen an, die denselben Effekt wie die Deklaration `open ISet` erzielen.

4. Deklarieren Sie mithilfe der Struktur `IMap` eine Struktur `ISet`, die endliche Mengen gemäß der Signatur `ISET` implementiert (siehe Abschnitt 9.2).

5. Deklarieren Sie einen Funktor `Map`, der zu den Argumenten

```
type key
val  compare : key * key -> order
```

eine Struktur liefert, die endliche Funktionen implementiert.

6. Deklarieren Sie mithilfe des Funktors `Map` eine Struktur `IMap`, die endliche Funktionen für *key* = *int* implementiert.

Aufgabe 9.3 (Priorisierte Schlangen) In Abschnitt 9.3 haben Sie gelernt, was man unter Schlangen versteht und wie man sie implementiert. Sie sollen jetzt sogenannte *priorisierte Schlangen* implementieren, deren Einträge Paare (k, v) sind, die aus einem *Schlüssel k* und einem Wert v bestehen. Für die Menge der Schlüssel muss eine Ordnung vorgegeben sein. Die Einträge einer priorisierten Schlange werden lexikografisch nach der Ordnung der Schlüssel und (für gleiche Schlüssel) nach dem Alter der Einträge geordnet. Das erste Element einer nichtleeren priorisierten Schlange ist also der Eintrag mit dem kleinsten Schlüssel, der schon am längsten in der Schlange ist. Die Einträge

```
(4, "Jim"), (2, "Maria"), (3, "Monica"), (2, "Tom")
```

sollen also die priorisierte Schlange

```
[(2, "Maria"), (2, "Tom"), (3, "Monica"), (4, "Jim")]
```

ergeben, wenn sie in der Reihefolge von links nach rechts erfolgen.

Für einen gegeben Typ *key* sollen priorisierte Schlangen gemäß der folgenden Signatur implementiert werden:

```
type 'a pqueue
val empty  : 'a pqueue
val insert : key * 'a * 'a pqueue -> 'a pqueue
val head   : 'a pqueue -> key * 'a  (* Empty *)
val tail   : 'a pqueue -> 'a pqueue     (* Empty *)
```

1. Deklarieren Sie eine Struktur `IPQueue`, die priorisierte Schlangen für *key* = *int* implementiert. Deklarieren Sie dazu zunächst eine passende Signatur `IPQUEUE`.

2. Deklarieren Sie einen Bezeichner

   ```
   val q : string IPQueue.pqueue
   ```

   der an die folgende Schlange gebunden ist:

   ```
   [(2, "Maria"), (2, "Tom"), (3, "Monica"), (4, "Jim")]
   ```

3. Deklarieren Sie einen Funktor `PQueue`, der für zwei Argumente

   ```
   type key
   val  compare : key * key -> order
   ```

eine Struktur liefert, die priorisierte Schlangen implementiert.

4. Deklarieren Sie mithilfe des Funktors `PQueue` eine Struktur `IPQueue`, die priorisierte Schlangen für *key* = *int* implementiert.