Programmiersprachen

Alexandru Berlea

Institut für Informatik
TU München

Wintersemester 2006/2007

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
```

Sharing Constraints

```
signature LexerSig = sig structure Symbol : SymbolSig
                         val next : unit -> Symbol.sym
                     end
signature SymTableSig =
  sig structure Symbol : SymbolSig
      structure Value : ValueSig
      type table
      val lookup : table * Symbol.sym -> Value.val
  end
functor Parser(Lexer:LexerSig, SymTable:SymTableSig) =
  struct ... SymTable.lookup(table,Lexer.next ()) ...end
```

- Allein unter Verwendung Informationen aus dem aktuellen Modul (der Argument-Signaturen), kann der Type-Checker nicht überprüfen, ob `SymTable.Symbol.sym` = `Lexer.Symbol.sym`.
- Unsere Absicht, dass `Lexer.Symbol` = `SymTable.Symbol` muss explizit dokumentiert werden.
- Zusätzliche Beziehungen zwischen Funktor-Argumenten muss man explizit mit Hilfe von **sharing constraints** angegeben.

Sharing Constraints für Typen...

... spezifizieren, dass zwei Typen identisch sein müssen:

```
signature LexerSig =
  sig structure Symbol : SymbolSig
      val next : unit -> Symbol.sym
  end

signature SymTableSig =
  sig structure Value : ValueSig
      structure Symbol : SymbolSig
      type table
      val lookup : table * Symbol.sym -> Value.val
      val update : table * Symbol.sym * Value.val ->table
  end

functor Parser(Lexer:LexerSig, SymTable:SymTableSig
               sharing type SymTable.Symbol.sym = Lexer.Symbol.sym) =
  struct ... SymTable.lookup(table,Lexer.next ()) ...end
```

- Bei der Funktor-Anwendung überprüft der Compiler, dass die zwei Typen gleich sind.

Sharing Constraints für Strukturen...

... spezifizieren, dass zwei Strukturen identisch sein müssen:

```
signature LexerSig =
  sig structure Symbol : SymbolSig
      val next : unit -> Symbol.sym
  end

signature SymTableSig =
  sig structure Value : ValueSig
      structure Symbol : SymbolSig
      type table
      val lookup : table * Symbol.sym -> Value.val
      val update : table * Symbol.sym * Value.val ->table
  end

functor Parser(Lexer:LexerSig, SymTable:SymTableSig
               sharing SymTable.Symbol = Lexer.Symbol) =
  struct ... SymTable.lookup(table,Lexer.next ()) ...end
```

Programmieren im Großen

- Eingabe längerer Programme auf der interaktiven Kommando-Zeile ist mühsam.

- $\Longrightarrow$ Lese sie aus Dateien ab
  - `CM.use : string -> unit` liest, übersetzt und macht den Inhalt einer Datei in der SML-Umgebung sichtbar.

```
- 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 und 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

- Um ein Projekt aus mehreren Dateien mit Hilfe von **use** zu kompilieren, müsste man:
  - sich die Datei-Namen merken;
  - die entsprechenden Folgen von `use`-Aufrufen verwalten.
- Dazu stellt SML/NJ die Struktur `CM` , der **Compilation Manager**, zur Verfügung.

Der Compilation Manager

- Zum Laden von Bibliotheken bietet CM:
  - `CM.make : unit -> unit` sucht eine Datei "sources.cm" und versucht, aus der dort angegebenen Spezifikation eine Bibliothek herzustellen.
  - `make' : string -> unit` benutzt stattdessen den angegebenen String als Datei-Namen.
- Bsp.-Spezifikation:

```
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").

Der Compilation Manager

- 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.

Der Compilation Manager

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

```
Library
    signature Bar
    structure Foo
is
    bar.sig
    foo.sml
```

Der Compilation Manager

- 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 einer Datei `utils.sml` und Datenstrukturen aus einer Datei `data.sml` zusammenfassen.

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

Der Compilation Manager

```
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
```

Der Compilation Manager

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.

Der Compilation Manager

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/NJ-Version oder Betriebssystem;
- Erzeugung von **Stand-alone-Versionen** (SMLofNJ-Struktur).

Überblick

3.
Eine nicht strikte funktionalle Sprache: Haskell

Nicht-strikte (verzögerte) Auswertung

Im Unterschied zu SML ist Funktionsauswertung in Haskell **nicht-strikt**, d.h. Argumente werden nur bei Bedarf (**verzögert**) ausgewertet. Betrachte die definition `f x = 1`. Dann liefert:

- In SML:

```
f (1 div 0)
uncaught exception divide by zero
```

- In Haskell:

```
f (1 `div` 0)
1 :: Integer
```

Basis-Typen in Haskell

Typ	Bsp.-Werte	Bsp.-Operatoren
Integer	`0 3 -7`	`+ div mod` : Integer $\times$ Integer $\mapsto$ Integer `-` : Integer $\mapsto$ Integer
Double	`-3.0 7.0`	`+ - * /` : Double $\times$ Double $\mapsto$ Double `~` : Double $\mapsto$ Double
Bool	`True False`	`not` : Bool $\mapsto$ Bool `||  &&` : Bool $\times$ Bool $\mapsto$ Bool
Char	`'a' 'b' '\n'`	

Funktionen

- Definition:

```
fact n = if n <= 0 then 1 else n * fact (n-1)
```

- Anwendung:

```
fact 4
24 :: Integer
```

Namenlose Funktionen

```
(\x -> x+1) 2
3 :: Integer
```

Produkt-Typen

- **Produkt-Konstruktor** wie in SML
- **Produkt-Typoperator** syntaktisch ähnlich wie der Produkt-Konstruktor:

```
(1,2)
(1,2) :: (Integer,Integer)
(True,'a')
(True,'a') :: (Bool,Char)
```

Summen-Typen

- Definition:

$$\begin{array}{l}\texttt{data T } \alpha_1\ \alpha_2\ \ldots\ \alpha_n = \mathit{Konstruktor}_1\ \beta_{11}\ \ldots\ \beta_{1n_1} \\ \qquad\qquad\qquad\qquad\quad \mid \mathit{Konstruktor}_2\ \beta_{21}\ \ldots\ \beta_{2n_2} \\ \qquad\qquad\qquad\qquad\quad \ldots \\ \qquad\qquad\qquad\qquad\quad \mid \mathit{Konstruktor}_m\ \beta_{m1}\ \ldots\ \beta_{mn_m}\end{array}$$

  Damit werden deklariert:

  - der **Typ-Operator** `T` (Prä-fixiert)
  - die **Konstruktor-Funktionen** (curried):

    $\mathit{Konstruktor}_i$ `::` β_{i1} `->` β_{i2} `->` `...` β_{in_1} `->` `T` α_1 `...` α_n

- Bsp.:
  - `data Tree a = Leaf a | Node (Tree a) a (Tree a)`

Pattern-Matching

- Ähnlich wie in SML:

```
case Ausdruck of
     Muster_1 -> Ausdruck_1
     Muster_2 -> Ausdruck_2
     ...
     Muster_n -> Ausdruck_n
```

- Bsp.:

```
nodes t =
  case t of
    Leaf _ -> 1
    Node t1 _ t2 -> 1 + (nodes t1) + (nodes t2)
```

- `if` e_1 `then` e_2 `else` e_3 ist eine spezielle Syntax für

```
case e_1 of True -> e_2
            False -> e_3
```

Listen

- Der Typ von Listen mit Elementen von Typ α, `[`α`]`, ist ein vordefinierter Summentyp mit:
  - Nullstelliger Konstruktor *nil*: `[]`
  - Einstelliger Konstruktor *cons*: `:` (rechtsassoziativ)
- Bsp.:
  - `1:2:3:[]`
  - Syntaktische Abkürzung: `[1,2,3]`
- Vordefinierte List-Funktionen: `head`, `tail`, `map`, `foldl`, `foldr`, `reverse`, `take`, `drop`, etc.

Strings

- `String` ist ein anderer Typ-Name (Synonym) für `[Char]`:

```
"hallo"
"hallo" :: String
reverse "hallo"
"ollah" :: [Char]
```

- Benutzer-definierte **Typ-Synonyme** werden mit `type` deklariert:

```
type Coordinate3D = (Integer, Integer, Integer)
```

Lokale Definitionen

- **Lokale Definitionen** werden mit dem `let`-Ausdruck eingeführt:

```
let  Definition_1
     Definition_2
     ...
     Definition_n
in   Ausdruck
```

  - *Definition*$_i$ ist der Form: *Pattern*$_i$ = *Ausdruck*$_i$
  - Die Definitionen sind in `Ausdruck` sichtbar;
  - Die Definitionen können verschränkt rekursiv sein;
  - Der Wert des `let`-Ausdrucks ist der Wert von *Ausdruck*.

```
fibo n =
  let f1 = 0
      f2 = 1
      fiboHelp f1 f2 n =
        if n<=0 then f1
        else fiboHelp f2 (f1+f2) (n-1)
  in fiboHelp f1 f2 n
```

Layout

- Keine Schlüsselwörter (wie etwa `val` oder `end` in SML) trennen die Definitionen in einem **let**-Ausdruck.
  $\Longrightarrow$ Sind folgende Definitionen äquivalent?

```
let  x = y
     z  t = x
in  z 1
```

```
let  x = y  z
     t = x
in  z 1
```

- Nein – Haskell verwendet zur Auflösung von Mehrdeutigkeiten beim Parsen die Einrückung (**Layout**) des Quellcodes:
  - Das Ende einer Deklaration wird durch ein nachfolgendes Symbol in einer Spalte <= Anfangsspalte der Deklaration signalisiert.

- Ähnliches gilt für die Trennung der Fälle in einem **case**-Ausdruck

Layout

- Das Layout ist eine verkürzte Syntax einer Einrückung-unabhängigen Spezifikation, die explizite Separator- und Terminator-Symbole benutzt, um einzelne Deklarationen voneinander abzugrenzen.

- Bsp.: alternative Syntax für **let**-Ausdrücke

```
let { Definition_1 ; Definition_1 ; ... ; Definition_n } in Ausdruck
```

- Die zwei Syntax-Formen können gemischt werden. Bsp.:

```
let x = 1; y = f
    g x = x
in g 1
```

Polymorphismus in Haskell

- Ähnlich wie SML unterstützt Haskell **parametrischer Polymorphismus**.
  Bsp. `length :: [a] -> Integer`
  - `length` hat eine einzige Definition
  - Keine Voraussetzungen an die Typen der Argumente $\Longrightarrow$ Typvariablen sind universal quantifiziert

- Zusätzlich unterstützt Haskell eine kontrollierte Form des **ad-hoc Polymorphismus** (*overloading*): der selbe Funktionsname steht für verschiedene Definitionen.
  - Bsp. Gleichheitstest-Fkt. `(==)` ist verschieden definiert für jeden verschiedenen Typ.
  - Andere Bsp.: `(+)`, `show`.

Typklassen

- Überladung stellt neue Herausforderungen an die statische Typ-Überprüfung
- **Problem**: **Typ-Überprüfung** – welchen Typ hat ==?
  - Man darf nicht den Typ einer überladenen Fkt. mit einer universell quantifizierten Typ-Variable parametrisieren.
- **Lösung** bei Haskell: assoziiere Funktionsnamen mit **Typ-Klassen**.

Typklassen

- **Syntax:** Typklasse = **Kollektion von Funktionsdeklarationen**.

```
class C α where
  f1 :: β1
  f2 :: β2
  ...
  fn :: βn
```

  - C = Typklassen-Name
  - α = Platzhalter für konkrete Typen (**Instanzen**)
  - β_1, ..., β_n = Typ-Ausdrücke, die α enthalten

- **Semantik:** Typklasse $C\ \alpha$ = Prädikat ((*type constraint/context*)), das die Existenz von (überladenen) Funktionsdefinitionen f_1, ..., f_n für eine Instanz α vorschreibt.

Typklassen

- Bsp.: Die Typklasse Eq ist wie folgt definiert:

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

- Als Effekt dieser Deklaration, wird == folgenden Typ zugewiesen:
  `(==) :: (Eq a) => a -> a -> Bool`
  Typ-Constraint `(Eq a)` beschränkt den polymorphen Typ a auf Instanzen der Klasse Eq

- Typ-Überprüfung propagiert Typ-Constraints:

```
isIn e l = case l of [] -> False
                     h:r -> (e == h) || isIn e r
isIn :: Eq a => a -> [a] -> Bool
```

Typinstanzen

- Die Zugehörigkeit eines Typs α zu einer Typklasse C wird wie folgt deklariert:

```
instance C α where
  f1 = Definition1
  f2 = Definition2
  ...
  fn = Definitionn
```

- Bsp.:

```
data State = On | Off

instance Eq State where
  (==) s1 s2 = case (s1,s2) of (On,On) -> True
                               (Off,Off) -> True
                               _   -> False
  (/=) s1 s2 = not (s1 == s2)
```

```
On == On
True :: Bool
```

Default-Definitionen

- Bei der Deklaration einer Klasse kann man **Default-Definitionen** spezifizieren:

```
class Eq a where
  (==) :: a -> a -> Bool
  x == y = not (x /= y)

  (/=) :: a -> a -> Bool
  x /= y = not (x == y)
```

- Jede Typ-Instanz besitzt diese Definition, wenn sie nicht überlädt

- Für `Eq` braucht man nur eine von `(==)` oder `(/=)` definieren.

Typ-Constraints in instance-Deklarationen

Bei instance-Deklarationen parametrisierter Typen müssen manchmal die Typ-Parameter eingeschränkt werden:

```
instance (Eq a) => Eq (Tree a) where
  (==) t1 t2 = case (t1,t2) of
     (Leaf x, Leaf y) -> (x==y)

     (Node tl1 k1 tr1, Node tl2 k2 tr2) ->
         (tl1==tl2) && (k1==k2) && (tr1==tr2)

     _ -> False
```