Programmiersprachen

Alexandru Berlea

Institut für Informatik
TU München

Wintersemester 2006/2007

Überblick

1. **Datentypen**
   - Einleitung
   - Vordefinierte Typen (Typ-Konstanten)
   - Definition neuer Typen (Typ-Ausdrücke)
   - Pattern-Matching
   - Rekursive Typen
   - Polymorphe Typen

Typen

- **Typen** = Mengen von Programm-Entitäten mit gleichartigem Verhalten

- Verwendung:
  - **Organisation**/Benennung von Konzepten, **Dokumentation**. Z.B.:
    - Basis-Typen: `int`,`float`
    - Komplexe Typen: `int channel` (CML), `int cont` (SML), Klassen (Java)

  - **Fehler erkennen und vermeiden** $\Longrightarrow$ **Type-safety**

  - **Optimierungen**: z.B. Komponente in einem Record/Objekt nachschlagen
    - Sequentielles durchsuchen, wenn der Typ unbekannt ist
    - Zugang via bekanntes Offset, sonst

Type-Safety

- **Type-Safety** = Programm-Entitäten gemäß ihrer Eigenschaften manipulieren $\Longrightarrow$ keine unbeabsichtigte Semantik

- erreicht durch **Type-Checking** = Überprüfung der Funktionsargumente auf Konformheit mit dem Typ des Definitionsbereichs
  - Der Typ jeder Entität muss bekannt sein, insbesondere für Definitions- und Bildbereich für Funktionen/Operatoren
  - keine Konformheit $\Longrightarrow$ Typ-Fehler, z.B. `1 + true`

Type-Safety

- Bsp.:
  - Type-safe: ML, Lisp, Java
  - Nicht type-safe: C, C++ $\Longrightarrow$ evt. unbeabsichtigte Semantik:
    - Implicit type-casts: ein Integer als Pointer benutzen
    - Pointer-Arithmetik: Wenn `p` Typ `T` hat, dann hat `p+1000` auch Typ `T`, obwohl die entsprechende Speicherzelle ein anderer Typ haben könnte.
    - Explizite Speicherfreigabe $\Longrightarrow$ evt. dangling pointers

Type-Checking

Type-Safety wird erreicht via **Type-Checking**

- **Laufzeit-Typechecking** (dynamisch): Compiler generiert Code zur Überprüfung, dass Operanden den richtigen Typ haben

  $\Longrightarrow$ Verlangsamung der Ausführung

- **Compilezeit-Typechecking** (static): die (meisten) Typ-Überprüfungen können bei modernen Programmiersprachen bei Compile-Zeit stattfinden.
  - **Weniger Programmierfehler**
  - **Laufzeit Robustheit**
  - **Optimierter Code**

Type-Checking

Type-Safety wird erreicht via **Type-Checking**

- **Laufzeit-Typechecking** (dynamisch): Compiler generiert Code zur Überprüfung, dass Operanden den richtigen Typ haben
  $\Longrightarrow$ Verlangsamung der Ausführung

- **Compilezeit-Typechecking** (static): die (meisten) Typ-Überprüfungen können bei modernen Programmiersprachen bei Compile-Zeit stattfinden.
  - **Weniger Programmierfehler**
  - **Laufzeit Robustheit**
  - **Optimierter Code**
  - aber notwendigerweise konservativ: ob ein Programm einen Typ-Fehler erzeugen könnte ist i.A. nicht entscheidbar:

```
if e then eWithTypeError else eWithTypeError
```

(stellt in der Praxis aber kein Problem:-)

Type-Checking

Laufzeit- und Compilezeit-Typechecking:
Die meisten Programmiersprachen benutzen eine Kombination der beiden, z.B. Java:

- statisch: Unterscheiden zwischen Ganzzahlen und Funktionen
- dynamisch: Array-Bounds check

Vordefinierte Typen in SML

Typ	Bsp.-Werte	Bsp.-Operatoren
int	`0 3 ~7`	`+ - * div mod` : int × int ↦ int `~` : int ↦ int
real	`3.0 7.0`	`+ - * /` : real × real ↦ real `~` : real ↦ real
bool	`true false`	`not` : bool ↦ bool `orelse andalso` : bool × bool ↦ bool
string	`"hallo"`	`^` : string × string ↦ string
char	`#"a" #"b"`	
unit	`()`	

Definition neuer Typen

- **Typ-Operatoren** konstruieren neue Typen aus bestehenden T.

$$op : MT^n \mapsto MT \text{ mit } n \geq 0$$

  mit MT die Menge der Typen in der Sprache.

- Die vordefinierten Basis-Typen (`real`,`int`,`bool`,`unit`) sind **nullstellige Typ-Operatoren/Typ-Konstanten**.

- Durch Anwendung der Typ-Operatoren enstehen **Typ-Ausdrücken**.

Produkt-Typen

Der Typ-Operator $* : MT^n \mapsto MT$ **mit** $n \geq 2$:

- steht zwischen Operanden (*infix operator*):
- ist definiert als:

$$\alpha_1 * \alpha_2 * \cdots * \alpha_n = \{(v_1, v_2, \ldots, v_n) \mid v_k \in \alpha_k \textbf{ für alle } k\}$$

Bsp.: $int * int = \{(x, y) \mid x, y \in int\}$

Produkt-Werte

Werte vom Produkt-Typ kann man mit Tupel-Wertekonstruktoren konstruieren

```
- (1,2);
val it = (1,2) : int * int

- (1,2,true);
val it = (1,2,true) : int * int * bool

- (1,(2,true));
val it = (1,(2,true)) : int * (int * bool)

- ((1,2),true);
val it = ((1,2),true) : (int * int) * bool
```

Die Symbolenkombination (,) kann man als einen verteilter Konstruktor auffassen.

Produkt-Werte

```
- (1,2,true) = (1,(2,true));
stdIn:42.1-42.26 Error: operator and operand don't agree [tycon mismatch]
 operator domain: (int * int * bool) * (int * int * bool)
 operand: (int * int * bool) * (int * (int * bool))
 in expression:
 (1,2,true) = (1,(2,true))

- (1,2,true) = (1,2,true);
val it = true : bool
```

Record-Typen (Verbunde)

Ein **Record-Typ** besteht aus Tupeln mit benannten Komponenten:

```
- val p1 ={vorName="John",name="Smith",alter="23"};
val p1 = {alter="23",name="Smith",vorName="John"}
: {alter:string, name:string, vorName:string}

- val p2 = {vorName="Jan",name="Smith"};
val p2 = {name="Smith",vorName="Jan"}
: {name:string, vorName:string}
```

- Zwei Record-Typen sind gleich wenn sie gleich viele Komponente haben, jeweils mit dem selben Namen und dem selben Typ:
- Zwei Record-Werte ($\equiv$ **Records**) sind gleich, wenn sie vom selben Typ sind, und die jeweiligen Komponenten gleich sind
- Die Symbolenkombination $\{,=\}$ kann man als einen verteilter Operator bzw. Wertekonstruktor auffassen.

Records

- Reihenfolge ist irrelevant

```
- {name="Schwarz",vorName="Peter",alter="25"} =
  {name="Schwarz",alter="25",vorName="Peter"};
val it = true : bool
```

- Tupel sind eine Spezialschreibweise für Records

```
- {1=true,2="Martin"};
val it = (true,"Martin") : bool * string
- {1=3,2=5};
val it = (3,5) : int * int
```

Summentypen

- **Summentyp** = eine Sammlung von Werten anderer Typen
  z.B. sollte der Typ **Publication** u.a. folgendes enthalten:
  - `{title="Generics",conf="Fun in the afternoon", auth="J. Smith"}:{auth:string, conf:string, title:string}`
  - `{title="ML in 7 days",publisher="Hupfer",auth="J. Valjean"}:{auth:string, publisher:string, title:string}`
  - `{title="P=NP",journal="JC",auth="J. Walker",edit="V. Smirnoff"}:{auth:string, edit:string, journal:string, title:string}`

  I.A. können diese Typen beliebig unterschiedlich sein...

- Lösung: **Summentyp** $S \in MT$ wird aus einer Menge von Typen $MT_1 \subset MT$ mit Hilfe von symbolischen Funktionen (**Konstruktoren**) c konstruiert:

$$S = \{c(v) \mid v \in \alpha, \alpha \in MT_1, c : \alpha \mapsto S\} \cup \{c \mid c : \bullet \mapsto S\}$$

Summentypen

- Ein Summentyp wird definiert mit **datatype** durch Angabe seiner **Konstruktoren**:

  `datatype` = $Konstruktor_1$ | $Konstruktor_1$ | ... | $Konstruktor_n$

  z.B.:

```
datatype Publication =
  Article of {title:string,conf:string,auth:string} |
  Book of {title:string,publisher:string,auth:string} |
  Comm of {title:string,journal:string,auth:string,
           edit:string} |
  Bible | Koran
```

- Konstruktoren können sein:
  - **einstellig**: Article, Book, Comm
  - **nullstellig**: Bible, Koran

Summentypen

- Noch ein Bsp.:

```
datatype Farbe = Rot | Blau | RGB of (int*int*int)
```

$$\texttt{Farbe} = \{\texttt{Rot}\} \cup \{\texttt{Blau}\} \cup \{\texttt{RGB (x,y,z)} \mid x, y, z \in \texttt{int}\}$$

- Der Compiler erkennt den Typ eines Wertes anhand des Konstruktors:

```
- Rot;
val it = Rot : Farbe
- RGB (80,200,130);
val it = RGB (80,200,130) : Farbe
```

Aufzählungstypen (enumeration types)

- **Aufzählungstyp** = Ein Summentypen mit nur nullstelligen Konstruktoren
- für endliche Typen, deren Wert **aufgezäht** werden können:

```
- datatype Farbe = Karo | Herz | Pik | Kreuz
= datatype Wert = Zehn | Bube | Dame | Koenig | As;
datatype Farbe = Herz | Karo | Kreuz | Pik
datatype Wert = As | Bube | Dame | Koenig | Zehn
```

Aufzählungstypen

Der Compiler erkennt den Typ eines Wertes anhand des Konstruktors:

```
- datatype Farbe = Karo | Herz | Pik | Kreuz
= datatype Wert = Neun | Zehn | Bube | Dame | Koenig | As;
datatype Farbe = Herz | Karo | Kreuz | Pik
datatype Wert = As | Bube | Dame | Koenig | Neun | Zehn
- Kreuz;
val Kreuz : Farbe
- val pik_bube = (Pik,Bube);
val pik_bube = (Pik,Bube) : Farbe * Wert
```

Aufzählungstypen vs. ad-hoc Kodierungen

Mögliche Kodierung: Benutze Paare von Strings und Zahlen, z.B.:

`("Karo","10")`	$\equiv$	Karo Zehn
`("Kreuz","12")`	$\equiv$	Kreuz Bube
`("Pik","1")`	$\equiv$	Pik As

Nachteile:

- Beim Test auf eine Farbe muß immer ein String-Vergleich stattfinden $\longrightarrow$ ineffizient
- Darstellung des Buben als 12 ist nicht intuitiv $\longrightarrow$ unleserliches Programm
- Welche Karte repräsentiert das Paar `("Kaor","1")`? (Tippfehler werden vom Compiler nicht bemerkt)

Vorteile Aufzählungstypen vs. ad-hoc Kodierungen

```
- datatype Farbe = Karo | Herz | Pik | Kreuz
= datatype Wert = Neun | Zehn | Bube | Dame | Koenig | As;
datatype Farbe = Herz | Karo | Kreuz | Pik
datatype Wert = As | Bube | Dame | Koenig | Neun | Zehn
```

Vorteile:

- Darstellung ist intuitiv.
- Tippfehler werden erkannt:

```
- (Kaor, As);
stdIn:29.2-29.6 Error: unbound variable or constructor: Kaor
```

- Interne Repräsentation ist effizient.

Aufzählungstypen vs. Basis-Typen

Manche Basis-Typen können als spezielle vordefinierte Aufzählungstypen aufgefasst werden:

- `datatype bool = true | false`
- `datatype char = #"a" | #"b" | #"c" | ...`
- `datatype int = ... | ~2 | ~1 | 0 | 1 | 2 | ...`

Pattern-Matching

Werte eines selben Typs können unterschiedlich behandelt werden, je nachdem mit welchem Konstruktor sie erzeugt wurden $\Longrightarrow$

Fall-Unterscheidung (*pattern matching*)

Die Fall-Unterscheidung erfolgt mit Hilfe des **case**-Ausdruckes:

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

Pattern-Matching: Beispiel

```
datatype Farbe = Rot | Blau | RGB of (int*int*int)
```

```
- val f = RGB (80,200,130);
val f = RGB (80,200,130) : Farbe
- val description = case f of
                      Rot => "pure red"
                    | Blau => "pure blue"
                    | RGB(80,200,130) => "scarlet"

val description = "scarlet" : string
```

- Im Unterschied zum gleichnamigen imperativen Konstrukt ist case einen Ausdruck (d.h. er hat einen Wert)
- In FP gibt es nicht den Unterschied zwischen Anweisungen und Ausdrücken $\Longrightarrow$ Alles ist ein Ausdruck
- Die rechten Seiten müssen alle den selben Typ haben; das ist der Typ des Gesamtausdruckes.

Der If-Ausdruck

- Oft findet die Fallunterscheidung über einen Wert vom Typ `bool`:

```
fun max (x,y) = case x>= y of true => x
                            | false => y;
val max = fn : int * int -> int

max (3,2);
val it = 3 : int
```

- Dafür gibt es eine alternative, kürzere Syntax:

```
val fun max (x,y) = if x>= y then x
                    else y;
val max = fn : int * int -> int
```

Pattern-Matching mit Variablen-Bindung

- Patterns können benutzt werden, um auf Bestandteile eines *konstruierten* Wertes zuzugreifen $\Longrightarrow$ **Dekomposition**
- Dafür müssen Patterns Variablen enthalten
- Beim Pattern-Matching über einen Wert werden die Variablen automatisch zu den entsprechenden Teilen des Wertes gebunden
- Die im Muster $Muster_i$ gebundenen Variablen sind im Ausdruck $Ausdruck_i$ sichtbar

```
- val f = RGB (80,200,130);
val f = RGB (80,200,130) : Farbe
- val description = case f of Rot => "pure red"
                            | Blau => "pure blue"
                            | RGB(x,y,z) =>
                        if (x=y) andalso (y=z) then "gray"
                        else "something else";
val description = "something else" : string
```

Pattern-Matching mit Variablen-Bindung

Wenn man eine Variablen-Bindung nicht braucht kann man _ (Unterstrich) benutzten

```
- val f = RGB (80,200,130);
val f = RGB (80,200,130) : Farbe
- val description = case f of
                      Rot => "pure red"
                    | Blau => "pure blue"
                    | RGB(x,y,_) =>
                      if (x=y) then "kind of yellow"
                      else "something else";
val description = "something else" : string
```

Pattern-Matching mit Variablen-Bindung

- ist ein wichtiges Feature, die die Dekomposition eines Wertes in seine Teile unterstützt
- Zusätzlich überprüft der Compiler automatisch, ob die Patterns redundant oder unvollständig sind...

Unvollständige Patterns

```
- val description = case f of Rot => "pure red"
                    | Blau => "pure blue"
                    | RGB(80,200,130) => "kind of green" ;
stdIn:78.13-108.57 Warning: match nonexhaustive
 Rot => ...
 Blau=> ...
 RGB (80,200,130) => ...
val description = "kind of green" : string
```

Alle Fälle sollten behandelt werden, evt. ähnlich wie unten:

```
- val description = case f of Rot => "pure red"
                    | Blau => "pure blue"
                    | RGB(80,200,130) => "kind of green"
                    | _ => "don't know";
val description = "kind of green" : string
```

Redundante Patterns

```
-   val description = case f of
                        Rot => "pure red"
                      | Blau => "pure blue"
                      | RGB(x,y,z) => "RGB colour"
                      | RGB(80,200,130) => "kind of green";
stdIn:125.8-139.57 Error: match redundant
 Rot => ...
 Blau => ...
 RGB (x,y,z) => ...
 --> RGB (80,200,130) => ...
```

Redundante Patterns

Achtung: Die Reihenfolge ist wichtig

```
- val description = case f of
                      Rot => "pure red"
                    | Blau => "pure blue"
                    | RGB(80,200,130) => "kind of green";
                    | RGB(x,y,z) => "RGB colour"
val description = "kind of green" : string
```

Pattern-Matching: Einschränkungen

- Patterns dürfen nur Konstruktoren enthalten:

```
case "abc" of prefix^suffix => prefix;
stdIn:66.1-66.38 Error: non-constructor applied to argument in pattern:^
stdIn:66.32-66.38 Error: unbound variable or constructor: prefix
```

$\Longrightarrow$ Eindeutigkeit:

- Höchstens eine Variable mit einem gegeben Namen in einem Pattern (**Linearität**):

```
- val description = case f of
                      Rot => "pure red"
                    | Blau => "pure blue"
                    | RGB(x,x,_) => "yellow"
stdIn: Error: duplicate variable in pattern(s): x
```

Rekursive Typen

Um beliebig große Datenstrukturen repräsentieren zu können braucht man **rekursive Typen** (z.B. Listen/Bäume).

- Die Definition eines Typen ist rekursiv, wenn Typ-Konstruktoren Elemente des zu definierenden Typ erhalten.

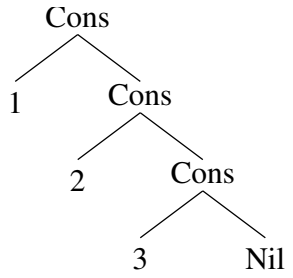
```
datatype IntList = Nil | Cons of (int*IntList);
```

- Damit Werte konstruierbar sind, muss midestens ein **nullstelliger Konstruktor** angegeben werden.
  Bsp.: Liste mit Elementen 1, 2, 3

```
Cons(1,
=       Cons(2,
=            Cons(3,Nil)));
val it = Cons (1,Cons (2,Cons (3,nil))) : IntList
```

Aufbau einer Liste

```
Cons(1,Cons(2,Cons(3,Nil)));
val it = Cons (1,Cons (2,Cons (3,nil))) : IntList
```



Verarbeitung rekursiver Datentypen...

... erfolgt mit Hilfe von Pattern-Matching und rekursive Funktionen
Die Länge einer Liste:

```
fun length l = case l of
                    Nil => 0
                  | Cons(first,rest) => 1 + length rest;
val length = fn : IntList -> int

length (Cons(1,Cons(2,Cons(3,Nil))));
val it = 3 : int
```

Polymorphe Typen

Listentypen unterscheiden sich nur in dem Typ ihrer Elemente:

```
datatype IntList = Nil | Cons of (int * IntList);
datatype RealList = NilR | ConsR of (real * RealList);
```

$\Longrightarrow$ manche Funktionen auf Listen sehen im Prinzip gleich aus:

```
fun lengthR l = case l of
                  NilR => 0
                | ConsR(first,rest) => 1 + length rest;
```

- Um Code-Duplizierung zu vermeiden, erlaube **polymorphe Funktionen**: statt nur Argumente eines bestimmten Typs, akkzeptiere Argumente verschiedener Typen.

- Wenn der Typ erlaubter Argumente eine Instanz eines Typ-Ausdruckes mit Typ-Variablen sein muss$\Longrightarrow$ **parametrischer Polymorphismus**.

Parametrischer Polymorphismus

- Ein **parametrisierter Typ** definieren:

```
datatype 'a List = Nil | Cons of ('a * 'a List);
datatype 'a List = Cons of 'a * 'a List | Nil
```

  - 'a ist ein Bezeichner für einen bestimmten beliebigen Typ (Typ-Variable), der in Typ-Ausdrücken in Konstruktoren benutzt werden darf.

- Der Compiler erkennt den Parameter-Typ automatisch:

```
- Cons(1,Cons(2,Cons(3,Nil)));
val it = Cons (1,Cons (2,Cons 3)) : int List
- Cons(1.0,Cons(2.0,Cons(3.0,Nil)));
val it = Cons (1.0,Cons (2.0,Cons 3.0)) : real List
- Cons(true,Cons(true,Cons(false,Nil)));
val it = Cons (true,Cons (true,Cons false)) : bool List
```