Programmiersprachen – Typsysteme

Programmierparadigmen

Johannes Brauer

29. Oktober 2019

Ziele

- Klärung wichtiger Begriffe aus der Theorie der Programmiersprachen
  - Typ, Datentyp, Typsystem
- Funktionale Programmiersprachen
  - Clojure
  - Standard ML

Motivation

- Die Verwendung der Begriffe *Klasse*, *Typ* und *Objekt* ist in der Softwaretechnik – insbesondere im Kontext der objektorientierten Programmierung – allgegenwärtig.
- Was ist der Typ eines Objekts?
- Was ist der Unterschied zwischen einem Typ und einer Klasse?
- Was ist ein abstrakter Datentyp?
- Haben o.g. Begriffe etwas zu tun mit Begriffen wie *Typkompatibilität*, *Unterklasse*, *Untertyp*, *Modul*, *Schnittstelle*, *Vererbung*, *Polymorphie* ... ?

Was ist ein Datentyp?

Interpretation von Bitmustern

Ein 32 Bit langes Binärwort

$$01110100011110010111000001100101$$

Was bedeutet es?

- Integer-Zahl mit dem Dezimalwert 1954115685,
- Gleitkommazahl gemäß IEEE-Norm: $7,90504 \cdot 10^{31}$

- Folge von vier Zeichen (z. B. codiert in ASCII oder ISO 8859-1): `type`
- Ein Maschinenbefehl bestehend aus
  - einem 8 Bit langen Befehlscode und
  - einer 24 Bit langen Speicheradresse

Woher „weiß“ die Maschine, was gemeint ist?
Gibt es einen Unterschied zwischen Programmen und Daten?

Rechner arbeiten typenlos

- Für die Ausführung von Programmen sind Typinformationen nicht erforderlich.
- In einem Programm einen Ausdruck der Art `1954115685 + 'type'` zu verwenden, ist offensichtlich unsinnig, der Prozessor hätte „kein Problem“ damit.
- Für die bestimmungsgemäße Interpretation der Bitmuster muss der Programmierer selbst sorgen.
- Die Hardware ist ohne Weiteres in der Lage, eine Zeichenfolge mit einer Integer-Zahl zu addieren oder zu multiplizieren.
- Das wichtigste Ziel, das mit der Typisierung von Daten, d. h. mit der Bildung von *Datentypen*, verbunden ist, besteht nun genau darin, dass auf Bitmuster keine typfremden Operationen ausgeführt werden. Die Prozessor-Hardware ist dazu selbst nicht in der Lage. Das Problem muss also softwaretechnisch gelöst werden.

Begriffsklärung

- Ein und dasselbe Bitmuster kann verschiedene Arten von Daten repräsentieren.
- Wir sagen,
  - „ein Datum besitzt einen bestimmten Typ” oder
  - „ist von einem Typ”.
- Der Begriff Datentyp bezeichnet mehr als eine bestimmte Art von Daten: Er stellt im Kontext der Programmiersprachen ein eigenständiges Konzept dar.
- Dies wird z. B. daran deutlich, dass auch Dinge, die gar keine Daten sind, einen Datentyp besitzen können, z. B. Variablen in Programmiersprachen wie C, Pascal oder Java.
- Obwohl der Begriff *Datentyp* der spezifischere ist, ist es durchaus üblich, den Begriff *Typ* als Synonym zu verwenden.
- Auch Funktionen besitzen einen Typ. In diesem Zusammenhang von *Datentyp* zu sprechen, ist eher ungebräuchlich.
- Statt einer präzisen Definition des Begriffs *Datentyp* soll daher der Frage nachgegangen werden: Wozu dienen Datentypen?

Das ungetypte Universum

- Die Menge der Bitmuster in einem Arbeitsspeicher kann als „ungetyptes Universum" betrachtet werden. Man könnte auch sagen, es gibt nur einen einzigen Typ.
- Folge: Die Hardware kann unsinnige Operationen nicht verhindern.
  - Was ist das Ergebnis der Addition zweier Speicheradressen?
  - Die Verwendung der Summe für den Zugriff auf den Arbeitsspeicher kann – zu möglicherweise gefährlichen – Fehlfunktionen des Programms führen.
- Ein Beispiel für ein ungetyptes Universum aus dem Bereich der Programmiersprachen ist die Menge der *symbolischen Ausdrücke* (vgl. [McC60])
- Eine Form von S-Ausdrücken sind Listen. Die Liste

  ```
  (f 4 5)
  ```

  kann in einem Kontext einfach eine Liste bestehend aus einem Symbol und zwei Zahlen sein. In einem anderen Kontext bedeutet die Liste die Anwendung einer Funktion `f` auf die Argumente 4 und 5.

Zweck der Typisierung

- Ordnung in das „Chaos" der ungetypten Universen bringen.
- Die Anwendung unsinniger, fehlerhafter Operationen vermeiden.
- Die Qualität von Software verbessern.
- Den Nachweis der Korrektheit von Software ermöglichen.
- **Typkompatibilität**:
  - Ein Operator kann auf seine Operanden nur dann fehlerfrei angewendet werden, wenn diese den erwarteten Typ haben.
  - Die Anwendung einer Funktion ist nur dann sinnvoll möglich, wenn ihre Argumente typkompatibel zu den in der Funktionsdefinition spezifizierten Parametern sind.
  - Für den Wert einer Funktionsanwendung muss wiederum Typkompatibiliät im Kontext seiner Verwendung gelten.

Syntaktische vs. semantische Typkompatibiliät

- Auch die Verwendbarkeit von größeren Software-Bausteinen, wie Objekten, Komponenten, Web-Services etc. kann auf die Typkompatibilität der an der Schnittstelle dieser Bausteine angebotenen Dienste zurückgeführt werden.
- Der Nutzer (Klient) einer Komponente muss wissen, wie die Dienste in Anspruch genommen werden können und welches Ergebnis ein Dienst liefert.

- Die erste Frage bezieht sich auf die *syntaktische* Kompatibilität, d. h. welche Typbezeichner und Funktionssignaturen zur Verfügung stehen.
- Die zweite Frage betrifft die *semantische* Kompatibilität, d. h. liefern die Dienste auch die erwarteten Resultate, verhalten sich die Funktionen gemäß ihrer Spezifikation.
- In vielen Texten, die sich mit der Typkompatibilität befassen, wird nur der syntaktische Aspekt beleuchtet. Für die Gewährleistung der Korrektheit von Software ist die Herstellung semantischer Kompatibilität unerlässlich.

Anthony Simons ([Sim02]) beschreibt, wie syntaktische Inkompatibilität Ursache des Verlusts des Mars Climate Orbiter war, während die Ariane-5-Katastrophe auf die Nutzung einer alten, semantisch inkompatiblen Software-Komponente zurückzuführen war.

Probleme eines „syntaktischen“ Typbegriffs

- Zwei Schnittstellen – Worin unterscheiden sie sich?

```
public interface Queue {
      /* Schnittstelle eines Puffers nach dem
       * Warteschlangenmodell (FIFO-Prinzip)
       */
      public void put(Object element);
      public Object get();
      public void init();
  }

  public interface Stack {
      /* Schnittstelle eines Stacks (LIFO-Prinzip)
       */
      public void push(Object element);
      public Object pop();
      public void init();
  }
```

Vor- und Nachteile syntaktischer Typen

Vorteile
- Unterstützung einer modularen Programmentwicklung durch die Trennung von Schnittstelle und Implementierung
- Allein die durch die Definition eines Datentyps geschaffene Möglichkeit, diesen neuen Datentyp wie einen vordefinierten Typ verwenden zu können, ist unzweifelhaft von großem Nutzen (vgl. in diesem Zusammenhang auch [SL10]).

Nachteile Die Definition eines Datentyps ausschließlich über seine syntaktische Schnittstelle (d. h. durch seinen Namen und die Signaturen der Operationen, aber ohne Spezifikation ihrer Semantik) erlaubt es dem Programmierer aber

1. zu einer Schnittstellenspezifikation verschiedene, möglicherweise semantisch inkompatible Implementierungen anzugeben oder
2. zu verschiedenen Schnittstellenspezifikationen, die eine unterschiedliche Semantik intendieren, eine identische Implementierung anzugeben.

Java-Interface für einen Typ `Tuple` ... ... zur Verdeutlichung des Problems

- Java-Interface, das einen Typ `Tuple` für Objekte mit zwei Komponenten und einer Additionsoperation definiert:

```
public interface Tuple {
    Object first();
    Object second();
    Tuple add(Tuple p);}
```

Implementierung von `Tuple` durch `Point`

```
public class Point implements Tuple {
    private int x;
    private int y;

    public Point(int xCoord, int yCoord) {
        x = xCoord;
        y = yCoord;}

    public Object second() {return y;}
    public Object first() {return x;}
    public Tuple add(Tuple p) {
        return new Point((Integer) this.first()
                         + (Integer) p.first(),
                         (Integer) this.second()
                         + (Integer) p.second());}}
```

Implementierung von `Tuple` durch `Fraction`

```
public class Fraction implements Tuple {
    private int numer;
    private int denom;
    public Fraction(int numerator, int denominator) {
        numer = numerator;
        denom = denominator; }
    public Object first() {return numer;}
    public Object second() {return denom;}
    public Tuple add(Tuple p) {
        int num;
        int den;
        den = (Integer) this.second()
              * (Integer) p.second();
```

```
num = (Integer) this.first() * p.second()
      + (Integer) p.first() * this.second();
return new Fraction(num, den);}}
```

Bewertung der Implementierungen

- Die unterschiedliche Implementierung der Methode `add` in den Klassen `Point` und `Fraction` entspricht natürlich den Erwartungen hinsichtlich des Verhaltens von Punkten bzw. rationalen Zahlen.
- Die beiden Klassen `Point` und `Fraction` das Interface `Tuple` implementieren zu lassen, erscheint zwar „böswillig“ ...
  - ... aber es ist problematisch, dass z. B. die folgende Anweisungssequenz syntaktisch korrekt ist:

    ```
    p = new Point(2, 3); f = new Fraction(3, 6);
    s1 = f.add(p);
    s2 = p.add(f);
    ```

  - Die Variablen `p`, `f`, `s1` und `s2` seien vom deklarierten Typ `Tuple`.
  - **Achtung**: `s1.first()` und `s2.first()` liefern unterschiedliche Ergebnisse.

Schlussfolgerung

- An diesem Beispiel wird deutlich, dass es zumindest fragwürdig ist, bei der Definition eines Typs (hier durch das Interface `Tuple`), auf die Spezifikation der Semantik der Operationen zu verzichten.
- Betrachtet man hingegen die Semantik der Operationen als Bestandteil einer Typdefinition, wären die von den Klassendefinitionen `Point` und `Fraction` erzeugten Typen wegen der unterschiedlichen Implementierung der `add`-Methode nicht kompatibel, bzw. das Interface `Tuple` wäre unvollständig, da die Spezifikation der Operationssemantiken fehlt.
- An dieser Stelle ist der Einwand, dass Java-Interfaces nicht in erster Linie der Definition von (abstrakten) Datentypen dienen, sondern als Ersatz fehlender Mehrfachvererbung eine Form der Polymorphie unterstützen sollen, nicht unberechtigt. In diesem Fall ist unterschiedliches Verhalten beim Senden der gleichen Nachricht (z. B. `add`) gerade erwünscht.

Typsysteme

Prüfbare Redundanz

- Da die „Bewohner“ eines ungetypten Universums die für die Prüfung der Typkompatibiliät nötigen Informationen nicht bereitstellen können, muss einem Programm, dessen Bestandteile auf typgemäße Verwendung hin überprüft werden sollen, prüfbare Redundanz hinzugefügt werden.
- Zwei Möglichkeiten:

1. Bei der Deklaration von Variablen ist eine Typangabe erforderlich:

```
int i, j (* in Java, C *)
var i, j: integer (* in Pascal, Modula, Oberon *)
```

2. Der Typ von Programmobjekten kann aus ihrer Notation oder ihrer Verwendung abgeleitet werden (Typinferenz). Der Clojure-Ausdruck

```
(def pi 3.14159)
```

   erlaubt es dem Clojure-Compiler, aus der Schreibweise der Zahl zu ermitteln, dass sie vom Typ `Float` ist und dem Variablenbezeichner `pi` ebenfalls diesen Typ zuzuordnen. Fortan kann geprüft werden, ob eine Verwendung von `pi` typgemäß erfolgt.

Statische vs. dynamische Typisierung

- Vereinfacht gesagt:

  statisch: Typprüfung erfolgt zur Übersetzungszeit.

  dynamisch: Typprüfung erfolgt während der Ausführung des Programms.

- In der Industrie werden statische Sprachen (wie Java, C++ und C#) viel häufiger benutzt als dynamische (wie CLOS, Python, Self, Perl, Php oder Smalltalk).

- Andererseits haben gerade Java und C# gewisse typische dynamische Techniken adaptiert (Garbage-collection, Portabilität, Reflection).

- Weitere Konzepte dynamischer Sprachen gewinnen an Bedeutung: Metaprogrammierung, Mobilität, dynamische Rekonfigurierbarkeit und Verteilung.

Präzisierung der Terminologie

- Ein *strenges Typsystem* stellt jederzeit sicher, dass auf eine Variable bzw. Objekt nur der typgemäße Satz von Operationen angewendet werden kann.

- Ein *schwaches Typsystem* lässt auch typfremde Operationen gelegentlich zu. Typisches Beispiel: Pointerarithmetik in C.

- Die Unterscheidung ist für konkrete Programmiersprachen nicht immer eindeutig.

- Ein *statisches Typsystem* führt Typprüfung durch eine statische Analyse des Programmes durch. (Programm wird hierzu nicht ausgeführt.) Es weist alle Programme zurück, die einem bestimmten Satz von Typregeln nicht genügen.

- Ein *dynamisches Typsystem* prüft zur Laufzeit unmittelbar vor Anwendung die Zulässigkeit einer Operation.

- Programmiersprachen mit einem strengen aber dynamischen Typsystem sind **nicht zu verwechseln** mit schwach getypten Sprachen. Erstere verursachen Ausnahmen bei fehlerhaften Operationen, letztere lassen unsinnige Operationen mit nicht vorhersagbaren Folgewirkungen zu.
- Auch wichtig: *statisch* ist nicht notwendig gleichbedeutend mit zur Übersetzungszeit. Gegenbeispiel: Eine Java Virtual Machine führt statische Typprüfungen am Byte-Code während des Ladens einer Klasse durch.
- Ein *explizites Typsystem* verlangt für jede Variable die Angabe eines Typs.
- Ein *implizites Typsystem* verlangt nur die Deklaration des Namens einer Variablen. Der Typ wird aus der Verwendung abgeleitet. Beispielsweise kann aus `(def x 12)` abgeleitet werden, dass `x` vom Typ Number oder Integer ist.
- **Zusammenfassung der Typ-Terminologie:** Durch die Begriffe wird ein dreidimensionales, orthogonales Koordinatensystem aufgespannt mit den Skalen:
  - schwach – streng
  - statisch – dynamisch
  - explizit – implizit

Einordnung von Programmiersprachen

- Vorbemerkung: Die genannten Kategorien stellen Kontinua dar. So werden z. B. auch in Sprachen mit dynamischer Typprüfung statische Prüfungen vorgenommen und umgekehrt.
- C hat ein schwaches, statisches und explizites Typsystem.
- Java besitzt ein strenges, statisches und explizites Typsystem.
- Haskell und ML besitzen ein strenges, statisches und implizites Typsystem.
- Lisp, Clojure und Smalltalk haben ein strenges, dynamisches und implizites Typsystem.

Sind strenge Typsysteme besser als schwache?

- Die Attribute *streng* und *schwach* stellen keine Qualitätsmerkmale dar.
- Wichtig ist der Verwendungszweck einer Programmiersprache.
- Beispielsweise ist die Programmiersprache C für die Systemprogrammierung geschaffen worden. Arithmetische Operationen auf Speicheradressen sind in diesem Zusammenhang nichts Ungewöhnliches.
- Es ist indes durchaus fragwürdig, die Programmiersprache C für die gewöhnliche Anwendungsentwicklung einzusetzen und damit die Vorteile eines strengen Typsystems aufzugeben.

Sind statische Typsysteme besser als dynamische?

- Statische Typsysteme haben den Vorteil, dass bestimmte Fehler bereits vor Ablauf des Programms gefunden werden.
- Allerdings weist jedes bekannte statische, explizite Typsystem auch viele eigentlich korrekte Programme zurück.
- Beispiel: Das Programmfragment

```
...  var i, j, h: integer; x, y: real;
...  h := i; i := j; j := h
```

  vertauscht die Werte zweier Integer-Variablen unter Verwendung einer "Hilfsvariablen"`h`. Wollte man außerdem die Werte zweier Real-Variablen vertauschen, kann `h` nicht wieder verwendet werden, da die Zuweisung einer Real-Variablen an eine Integer-Variable

```
...  h := x; ...
```

  eine Typverletzung darstellt.
- Das Problem tritt in dieser Form nur in *monomorphen* Sprachen auf.

Sind implizite Typsysteme besser als explizite?

- Implizite Typsysteme befreien den Programmierer davon, redundante Typinformationen dem Programm hinzuzufügen.
- Implizite Typsysteme sind meist auch dynamisch (z. B. Smalltalk, Clojure).
- Es gibt aber auch Programmiersprachen, die ein statisches, implizites Typsystem besitzen (z. B. Haskell, ML).
- Explizite Typsysteme dokumentieren den Programmtext durch die Typangaben.
- Dieser Zusatznutzen wird jedoch wegen des erfoderlichen Zeitaufwands für den Programmierer infrage gestellt.

Für eine eingehendere Betrachtung des Themas „Typsysteme“ vgl. Kapitel 5 in [Bra09].

Funktionale Programmiersprachen

Clojure

Eigenschaften von Clojure

Clojure ...

- ist – wie Racket – ein Lisp-Dialekt.

- läuft auf der Java Virtual Machine.
- besitzt ein strenges, dynamisches, implizites Typsystem.
- ermöglicht verschiedene Programmierstile (imperativ, **funktional**, objektorientiert, ...).

Web-Seiten zu Clojure

- Hauptseite
- Features
- Download
- Dokumentation
  - clojuredocs
  - clojure-doc
- Google Gruppe
- Tutorials
  - Getting Started
  - Learn Clojure in 5 minutes
  - CLOJURE for the BRAVE and TRUE
  - Clojure - Functional Programming for the JVM

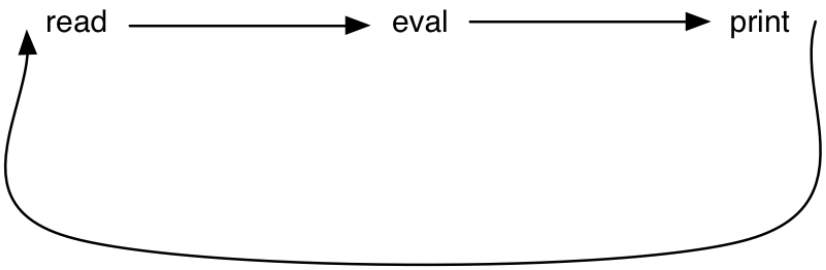
Benutzung von Clojure

- Online Clojure IDE
- Command line tool: `clj`
- Clooj - editor and IDE written entirely in Clojure itself
  - clooj, a lightweight IDE for clojure
  - Download
- LightTable is a new interactive IDE ...
- Leiningen
- La Clojure and IntelliJ IDEA
- Eclipse and Counterclockwise
- Clojure with Emacs

Die Read-Eval-Print-Loop

Funktioniert wie in Racket (Interaktionsfenster in DrRacket)

- Read: lies einen Ausdruck
- Eval(uate): werte ihn aus
- Print: gib das Ergebnis aus



Beispiele

Clojure

```
(def sum
      (fn [lvz]
        (cond
          (empty? lvz) 0
          :else (+ (first lvz) (sum (rest lvz))))))
```

Racket

```
(define sum
 (lambda [lvz]
   (cond
     [(empty? lvz) 0]
     [else (+ (first lvz) (sum (rest lvz)))])))
```

Clojure

```
(def filter
      (fn [praed? lst]
        (cond
          (empty? lst) ()
          (praed? (first lst))
             (cons (first lst) (filter praed? (rest lst)))
          :else (filter praed? (rest lst)))))
```

Racket

```
(define filter
  (lambda [praed? lst]
    (cond
      [(empty? lst) empty]
      [(praed? (first lst))
         (cons (first lst) (filter praed? (rest lst)))]
      [else (filter praed? (rest lst))])))
```

Standard ML

Eigenschaften von ML

ML ...

- ist eine funktionale Programmiersprache.
- besitzt eigene native Compiler (für verschiedene Plattformen).
- besitzt ein strenges, statisches, implizites Typsystem.
- arbeitet mit Typinferenz.
- ermöglicht Mustervergleich (pattern matching) anstelle von expliziten Fallunterscheidungen.
- ist eigentlich eine Sprachfamilie, zu der u. a. *Standard ML*, OCAml, F# und Elm gehören.

Wir benutzen Standard ML.

Web-Seiten zu ML (SML)

- Standard ML Family GitHub Project
- Standard ML of New Jersey
- ordentliche Zusammenfassung auf Wikipedia
- Bücher/Tutorials
  - Learn Standard ML in some minutes
  - Programming in Standard ML
  - Programming in Standard ML '97: A Tutorial Introduction
  - Introduction to SML
  - ML for the Working Programmer
  - Gert Smolka: Programmierung – eine Einführung in die Informatik mit Standard ML (zweite Auflage)

Benutzung von SML

- Online-Interpreter SOSML
- Using SML and Emacs oder SML/NJ and Emacs SML Mode

Beispiele

SML

```
fun sum (lvz : int list) =
    if null lvz
    then 0
    else hd lvz + sum(tl lvz)
```

Racket

```
(define sum
 (lambda [lvz]
   (cond
     [(empty? lvz) 0]
     [else (+ (first lvz) (sum (rest lvz)))])))
```

SML

```
fun filter (praed, lst) =
    if null lst
    then []
    else if praed (hd lst)
    then
        hd lst :: filter (praed, tl lst)
    else filter (praed, tl lst)
(* Signatur von filter: fn : ('a -> bool) * 'a list -> 'a list *)
```

Racket

```
(define filter
  (lambda [praed? lst]
    (cond
      [(empty? lst) empty]
      [(praed? (first lst))
         (cons (first lst) (filter praed? (rest lst)))]
      [else (filter praed? (rest lst))])))
```

SML – Kurzanleitung

Nutzung des Interpreters

Der SML/NJ-Interpreter wird von der Kommandozeile mit `sml` (oder besser: `rlwrap sml`)gestartet:

```
Standard ML of New Jersey v110.82 [built: Tue Jan  9 20:54:24 2018]
-
```

Das --Zeichen ist die Eingabeaufforderung. Hier können ML-Ausdrücke eingegeben werden:

```
- 3+4;
val it = 7 : int
- "abc"^"xyz"
= ;
val it = "abcxyz" : string
-
```

Einfache Deklarationen

- Erzeugen einer Deklaration (Variablenbindung):

```
- val x = 2*3+4;
val x = 10 : int
```

- `x` steht ab jetzt für 10.
- Benutzung:

```
- val y = x*2;
val y = 20 : int
```

- globale Umgebung `{x = 10, y = 20}`

Einfache Typen

• Neben ganzen Zahlen (`int`):

```
- 1.0;
val it = 1.0 : real
- ~3.32E~7;
val it = ~3.32E~07 : real
- "abc";
val it = "abc" : string
- #"a";
val it = #"a" : char
- 2 = 2;
val it = true : bool
- true;
val it = true : bool
- false;
val it = false : bool
```

Typen besitzen spezifische Operationen:

Typ	Operationen
int	~, +, *, -, div, mod, <, >, <=, >=, =
real	~, +, *, -, /, <, >, <=, >=
string	^
boolean	not, andalso, orelse, =

- Auch wenn einige Operationen für `int` und `real` anwendbar sind, dürfen Werte dieser beiden Typen nicht direkt verknüpft werden. Vorher ist eine Typanpassung erforderlich:

```
- 2 + 3.0;
stdIn:1.2-17.1 Error: operator and operand don't agree [overload conflict]
  operator domain: [+ ty] * [+ ty]
  operand:         [+ ty] * real
  in expression:
    2 + 3.0
- real 2 + 3.0;
val it = 5.0 : real
```

Funktionen

- Lambda-Ausdrücke:

```
- fn x => x+1;
val it = fn : int -> int
- (fn x => x+1) 5;
val it = 6 : int
```

- Benennung von Funktionen

```
- val twice = (fn x => 2*x);
val twice = fn : int -> int
- twice y;
val it = 40 : int
```

- Was tun bei rekursiven Funktionen?
- Man verwendet `val rec`:

```
- val rec fac = (fn  n => if (n=0) then 1 else n*(fac (n-1)));
val fac = fn : int -> int

- fac 5;
val it = 120 : int
```

- oder man verwendet die Funktionsdeklaration mit `fun`:

```
- fun fac n = if (n=0) then 1 else n*(fac (n-1));
val fac = fn : int -> int
```

- Das ist die gebräuchlichere Variante.

Funktionen verallgemeinert (vorläufig)

- Die vorläufige Syntax für eine Funktionsdeklaration:

  ```
  fun x0 (x1 : t1, ..., xn : tn) = e
  ```

- Bindung einer Funktion mit
  - dem Namen `x0` und
  - den Parametern `x1, ... xn`, deren
    - Typen `t1, ..., tn` sind, und
  - einer Berechnungsvorschrift `e`.
- Die runden Klammer um die Parameter sind nur bei mehr als einem Parameter erforderlich.
- Die Syntax eines Funktionstyps (die Signatur einer Funktion):

  ```
  ‘‘parameter types’’ -> ‘‘result type’’
  ```

  wobei die Parametertypen durch `*` voneinander getrennt werden.
- Beispiel:

  ```
  fun pow (x:int, y:int) = (* correct only for y >= 0 *)
      if y=0
      then 1
      else x * pow(x,y-1)

  val pow = fn : int * int -> int
  ```

Funktionsaufrufe

- Syntax

  `e0 (e1,...,en)`

  Runde Klammern sind nur bei mehr als einem Argument erforderlich.

- Typregeln:
  - `e0` muss einen Typ der Form `t1*...*tn->t` haben, wobei gilt: für alle $1 \leq i \leq n$, ei hat Typ ti.
  - Der Aufruf selbst hat den Typ `t`.

- Beispiel:

```
fun pow (x:int, y:int) = (* correct only for y >= 0 *)
    if y=0
    then 1
    else x * pow(x,y-1)
fun cube (x:int) =
    pow(x,3)
val ans = cube(4)

(*Auswertung:*)
val pow = fn : int * int -> int
val cube = fn : int -> int
val ans = 64 : int
```

Lokale Definitionen

- Syntax:

  `let b1 b2 ... bn in e end`

  mit `bi` ist Bindung, `e` ist Ausdruck

- Beispiel:

```
let val x = 1
in
    (let val x = 2 in x+1 end) + (let val y = x+2 in y+1 end)
end
val it = 7 : int
```

Optionale Werte (SML Options)

- Unter optionalen Werten (Options) versteht man solche, die existieren können aber nicht müssen:

  `NONE` ist ein `Option`-Wert, der „nichts enthält".

  `SOME e` ist ein `Option`-Wert, der den Wert von `e` enthält.

- Der Typ der Funktion

```
fun divide (x,y) = x div y
val divide = fn : int * int -> int
```

ist eigentlich „gelogen". Warum?

- Die Funktion

```
fun divide (x, y) = if y = 0 then NONE else SOME (x div y)
```

erzeugt niemals eine Exception.

Anwendungsbeispiel

- Die folgende Funktion erzeugt eine Exception für leere Listen:

```
fun max (xs : int list) =
    if null (tl xs)
    then hd xs
    else
        let val tl_ans = max(tl xs)
        in
            if hd xs > tl_ans
            then hd xs
            else tl_ans
        end
```

- bessere Variante:

```
fun better_max (xs : int list) =
    if null xs
    then NONE
    else let (* fine to assume argument nonempty because it is local *)
            fun max_nonempty (xs : int list) =
                if null (tl xs) (* xs must not be [] *)
                then hd xs
                else let val tl_ans = max_nonempty(tl xs)
                     in
                         if hd xs > tl_ans
                         then hd xs
                         else tl_ans
                     end
         in
            SOME (max_nonempty xs)
         end

- better_max [3,5,6,1];
val it = SOME 6 : int option
- better_max [];
val it = NONE : int option
```

```
- isSome (better_max []);
val it = false : bool
- isSome (better_max [3,4,5,1]);
val it = true : bool
- valOf (better_max [3,4,5,1]);
val it = 5 : int
- valOf (better_max []);
uncaught exception Option
```

Zusammengesetzte Typen: Tupel

- Paare und n-Tupel:

```
- val pair = (1,"abc");
val pair = (1,"abc") : int * string
- val triple = (1,true,1.0);
val triple = (1,true,1.0) : int * bool * real
- val nested = (7,(true,9));
val nested = (7,(true,9)) : int * (bool * int)
```

- Komponentenzugriff:

```
- #3(triple);
val it = 1.0 : real
- val (x,y) = pair;
val x = 1 : int
val y = "abc" : string
- #2(nested);
val it = (true,9) : bool * int
```

Funktionen über Tupel

Adhoc-Aufgabe: Schreiben Sie ein Java-Programm mit einer Methode `div_mod`, die ein Paar von `int`-Zahlen akzeptiert und ein Paar mit dem Wert der ganzzahligen Division und dem Divisionsrest zurückgibt.

- Die ML-Variante:

```
fun div_mod (x : int, y : int) =  (x div y, x mod y)
```

- die Clojure-Variante

```
(defn div_mod [[x y]] [(quot x y) (rem x y)])
```

Zusammengesetzte Typen: Listen

- Listenkonstruktion:

```
- 1::nil;
val it = [1] : int list
- val lst = [1,2,3];
val lst = [1,2,3] : int list
```

```
- 0::lst;
val it = [0,1,2,3] : int list
- nil;
val it = [] : 'a list
```

- Zugriffsoperationen:

```
- hd lst;
val it = 1 : int
- tl lst;
val it = [2,3] : int list
- tl(tl(tl lst));
val it = [] : int list (* [] steht für die leere Liste *)
```

Funktionen über Listen

- Beispiele:

```
fun sum_list (xs : int list) =
    if null xs
    then 0
    else hd(xs) + sum_list(tl xs)

fun append (xs : int list, ys : int list) =
    if null xs
    then ys
    else (hd xs) :: append(tl xs, ys)

(* Anwendungsbeispiele: *)
- sum_list [3, 4, 5];
val it = 12 : int
- append ([1,2,3] [4,5,6]);
val it = [1,2,3,4,5,6] : int list
```

Zusammengesetzte Typen: Verbünde (Records)

- Records haben die Form:

  `{lab1=exp1,···,labn=expn}`

- Beispiel:

  `{make = "Ford", built = 1904}`

- Records sind den Hashmaps von Clojure sehr ähnlich.
- Die Reihenfolge der Felder in den Records ist unerheblich.
- Zugriffsoperationen:

  ```
  - val car = {make = "Ford", built = 1904};
  val car = {built=1904,make="Ford"} : {built:int, make:string}
  - #make car;
  val it = "Ford" : string
  ```

- Paare und n-Tupel sind spezielle Records mit den Labels 1, 2, etc.

Der Typ `unit`

- Es gib einen eingebauten Typ `unit`.
- Es handelt sich um ein Alias für den Tuple-Typ `{}`.
- Der Typ besitzt nur ein Exemplar, das 0-Tupel `{}`, das auch so geschrieben wird: `()`.
- Dieser Wert wird häufig auch `unit` genannt.
- `unit` wird z. B. als Resultattyp für Funktionen benutzt, die eigentlich keinen Wert zurückgeben, sondern nur einen Effekt haben.

Literaturverzeichnis

[Bra09] Johannes Brauer. Typen, Objekte, Klassen – Teil 1: Grundlagen. Arbeitspapier 2009-05, NORDAKADEMIE Hochschule der Wirtschaft, Juni 2009.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, April 1960.

[Sim02] Anthony J. H. Simons. The theory of classification, part 1: Perspectives on type compatibility. *Journal of Object Technology*, 1(1):55–61, May-June 2002.

[SL10] Paolo A.G. Sivilotti and Matthew Lang. Interfaces first (and foremost) with java. In *Proceedings of the 41st ACM technical symposium on Computer science education*, SIGCSE '10, pages 515–519, New York, NY, USA, 2010. ACM.