Ludwig Maximilians Universität München
Institut für Informatik
Lehr- und Forschungseinheit für Datenbanksysteme

Skript zur Vorlesung
Informatik I
Wintersemester 2006

Kapitel 8: Abstraktionsbildung mit neuen Typen

Vorlesung: Prof. Dr. Christian Böhm
Übungen: Elke Achtert, Arthur Zimek

Skript © 2006 Christian Böhm

http://www.dbs.ifi.lmu.de/Lehre/Info1

Database Systems

Inhalt

1. Typen im Überblick
2. Deklarationen von Typabkürzungen in SML: type-Deklarationen
3. Definition von Typen: datatype-Deklarationen
4. Definition von rekursiven Typen
5. Beweisprinzip der strukturellen Induktion
6. Beispiele: Implementierungen des Konzepts der „Menge“
7. Beispiele: Grundlegende Funktionen für binäre (Such-) Bäume

Überblick

1. Typen im Überblick
2. Deklarationen von Typabkürzungen in SML: type-Deklarationen
3. Definition von Typen: datatype-Deklarationen
4. Definition von rekursiven Typen
5. Beweisprinzip der strukturellen Induktion
6. Beispiele: Implementierungen des Konzepts der „Menge“
7. Beispiele: Grundlegende Funktionen für binäre (Such-) Bäume

Typ als Wertemenge

- Ein Typ (oder Datentyp) bezeichnet eine Menge von Werten.
- Diese Werte können atomar (z.B. die Werte des Typs „ganze Zahlen“: `int`) oder zusammengesetzt (z.B. die Werte des Typs „Listen von ganzen Zahlen“: `int list`) sein.
- Die Wertemenge, die ein Typ repräsentiert, kann endlich (z.B. im Falle des Typs „Boole‘sche Werte“: `bool`) oder unendlich (wie im Falle des Typs „ganze Zahlen“: `int`) sein.
- Mit einem Typ werden üblicherweise Prozeduren zur Bearbeitung der Daten des Typs angeboten.

Typen mit atomaren und zusammengesetzten Werten

- Ein Typ kann atomare Werte haben (z.B. die Typen `bool` und `int`).
- Ein Typ kann auch zusammengesetzte Werte haben (z.B. die Typen `int * int` und der Typ `'a list`).
- Typen mit zusammengesetzten Werten werden auch deshalb so genannt, weil sie mit zusammengesetzten Typausdrücken wie `real * real` oder `int list` oder `'a list` bezeichnet werden.

Typen in Programmiersprachen mit erweiterbaren Typsystemen

- Ein Typ kann vordefiniert sein, d.h. von der Programmiersprache angeboten werden.
- Vordefinierte Typen in SML sind z.B. `int` und `bool`.
- Moderne Programmiersprachen ermöglichen auch die Definition von neuen Typen nach den Anforderungen einer Programmieraufgabe.
- Diese Sprachen haben ein „*erweiterbares Typsystem*“.
- Programmiersprachen mit erweiterbaren Typsystemen ermöglichen z.B. die Definition eines Typs „Wochentag“, eines Typs „Uhrzeit“, eines Typs „komplexe Zahl“ oder auch eines Typs „Übungsgruppe“ jeweils mit einer geeigneten Wertemenge (vgl. Kapitel 5).

Monomorphe und Polymorphe Typen

- Ein (atomarer oder zusammengesetzter) Typ kann *monomorph* sein:

  Dann kommt keine Typvariable im Typausdruck vor, der diesen Typ bezeichnet.
- Beispiel: `bool`

  `int * int`
- Ein (atomarer oder zusammengesetzter) Typ kann auch *polymorph* sein:

  Dann kommen eine oder mehrere Typvariablen im zugehörigen Typausdruck vor.
- Beispiel: `'a list`

(Wert-) Konstruktoren eines Typs

- Zusammen mit einem Typ werden (Wert-)Konstruktoren definiert.
- Der vordefinierte (polymorphe) Typ „Liste“ hat z.B. zwei (Wert-) Konstruktoren:
  - die leere Liste `nil`
  - den Operator `cons (::)`
- (Wert-) Konstruktoren können 0-stellig sein (Konstanten) oder eine beliebige andere Stelligkeit haben.
- Der (Wert-) Konstruktor `nil` des polymorphen Typs „Liste“ ist 0-stellig.
- Der (Wert-) Konstruktor `cons (::)` desselben Typs ist zweistellig (binär).

(Wert-) Selektoren eines Typs

- Für zusammengesetzte Typen werden auch „(*Wert-) Selektoren*“ definiert, womit die zusammengesetzten Werte des Typs zerlegt werden können.
- Die (Wert-) Selektoren des vordefinierten (polymorphen) Typs „Liste“ sind die (vordefinierten) Funktionen:
  - `hd` (*head*)
  - `tl` (*tail*)
- (Wert-) Selektoren werden auch „*Destruktoren*“ genannt.

Typkonstruktoren

- Zur Definition von Typen werden Typkonstruktoren angeboten, mit denen Typausdrücke zusammengesetzt werden können (Kapitel 6.5).
- Typkonstruktoren unterscheiden sich syntaktisch nicht von Funktionen.
- Typkonstruktoren werden aber anders als Funktionsnamen verwendet:
  - Wird eine Funktion auf aktuelle Parameter angewandt, so geschieht dies, um einen Wert zu berechnen.
  - Wird ein Typkonstruktor auf Typausdrücke angewandt, so geschieht dies, um einen neuen Typausdruck zu bilden und so einen neuen Typ zu definieren. Dabei findet keine Auswertung (im Sinne des Auswertungsalgorithmus aus Kapitel 3.1) statt.

Überblick

1. Typen im Überblick
2. Deklarationen von Typabkürzungen in SML: type-Deklarationen
3. Definition von Typen: datatype-Deklarationen
4. Definition von rekursiven Typen
5. Beweisprinzip der strukturellen Induktion
6. Beispiele: Implementierungen des Konzepts der „Menge“
7. Beispiele: Grundlegende Funktionen für binäre (Such-) Bäume

Erinnerung an das Vektor – Beispiel aus Kapitel 5.3

```
- type punkt = real * real;
type punkt = real * real

- fun abstand(p1: punkt, p2: punkt) =
            let fun quadrat(z) = z * z
                  val delta_x = #1(p2) - #1(p1)
                  val delta_y = #2(p2) - #2(p1)
            in
                  Math.sqrt(quadrat(delta_x)+
                  quadrat(delta_y))
            end;
val abstand = fn : punkt * punkt -> real

- abstand((4.5, 2.2), (1.5, 1.9));
val it = 3.01496268634 : real
```

Typabkürzungen

- Mit der type-Deklaration
  - `type punkt = real * real;`

  wird die Typkonstante `punkt` als Abkürzung für den Vektortyp `real * real` vereinbart.
- So eine Abkürzung wird „*Typabkürzung*“ (*type abbreviation*) genannt.
- Sie spezifiziert keinen neuen Typ, sondern ein Synonym für einen bereits vorhandenen Typ.
- Bietet eine Programmiersprache Typabkürzungen, so sagt man auch, dass die Programmiersprache eine „*transparente Typbindung*“ (*transparent type binding*) ermöglicht.

Grenzen der Nutzung von Typabkürzungen (1)

- Beispiel:

  Benötigt man neben Kartesischen Koordinaten auch Polarkoordinaten, kann man folgende Typabkürzungen vereinbaren:

```
- type kartes_punkt = real * real;
type kartes_punkt = real * real

- type polar_punkt = real * real;
type polar_punkt = real * real
```

Grenzen der Nutzung von Typabkürzungen (2)

- Da `punkt` und `kartes_punkt` beide denselben Typ `real * real` bezeichnen, ist keine Anpassung der Definition der Funktion `abstand` an die neu eingeführte Typabkürzung `kartes_punkt` nötig:

```
- val A = (0.0, 0.0) : kartes_punkt;
val A = (0.0,0.0) : kartes_punkt

- val B = (1.0, 1.0) : kartes_punkt;
val B = (1.0,1.0) : kartes_punkt

- abstand (A, B);
val it = 1.41421356237 : real
```

- Das ist zwar bequem, aber auch gefährlich, weil die Funktion `abstand` auch auf Punkte in Polarkoordinaten angewendet werden kann.

Grenzen der Nutzung von Typabkürzungen (3)

- Aus der Sicht der Anwendung macht es keinen Sinn `abstand` auf Polarkoordinaten anzuwenden:

```
- val C = (1.0, Math.pi/2.0) : polar_punkt;
val C = (1.0,1.57079632679) : polar_punkt

- abstand(B, C);
val it = 0.570796326795 : real
```

- Der Punkt `C` hat die Kartesischen Koordinaten `(0.0, 1.0)`, der Abstand zwischen `B` und `C` ist also `1.0`.

Nützlichkeit von Typabkürzungen: 1. Beispiel

- Betrachte die Funktionsdeklaration `abstand`.
- Ohne Typabkürzungen müsste `abstand` so definiert werden:

```
- fun abstand(p1: real * real, p2: real * real) =
            let   fun quadrat(z) = z * z
                  val delta_x = #1(p2) - #1(p1)
                  val delta_y = #2(p2) - #2(p1)
            in
                  Math.sqrt(quadrat(delta_x) +
                  quadrat(delta_y))
            end;
val abstand = fn : (real * real) * (real * real) -> real
```


Vergleich der Lesbarkeit

- Besonders der ermittelte Typ
  `(real * real) * (real * real) -> real`
  der Funktion `abstand` ist wesentlich schlechter lesbar als
  `punkt * punkt -> real`.

Nützlichkeit von Typabkürzungen: 2. Beispiel (1)

```
- type person = {Name:string,Vorname:string,
                 Anschrift:string,Email:string,Tel:int};
type person = {Anschrift:string,Email:string,
               Name:string, Tel:int, Vorname:string}

- type kurz_person = {Name:char,Vorname:char,Tel:int};
type kurz_person = {Name:char,Tel:int,Vorname:char}

- fun kuerzen(x:person) : kurz_person =
           {Name = String.sub(#Name(x),0),
            Vorname = String.sub(#Vorname(x),0),
            Tel = #Tel(x)};
val kuerzen = fn : person -> kurz_person
```


Nützlichkeit von Typabkürzungen: 2. Beispiel (2)

- Obwohl die Typausdrücke

  `person -> kurz_person`

  und

  `{Anschrift:string,Email:string,Name:string,Tel:int, Vorname:string} -> {Name:char,Tel:int,Vorname:char}`

  denselben Typ bezeichnen, ist der erste Ausdruck besser lesbar.

- Noch ein Beispiel für die Verwendung der Funktion:

```
- kuerzen
  {Name="Meier", Vorname="Franz", Anschrift="Muenchen",
  Tel=2210, Email="meier@bayern.de"};
val it = {Name=#"M",Tel=2210,Vorname=#"F"} : kurz_person
```

Polymorphe Typabkürzungen

- Typvariablen dürfen in type-Deklarationen vorkommen.
- Die Typabkürzungen, die so vereinbart werden, heißen „*polymorphe Typabkürzungen*“.
- type-Deklarationen, in denen Typvariablen vorkommen, heißen „*parametrische type-Deklarationen*“.

- Beispiel:

```
- type 'a menge = 'a -> bool;
type 'a menge = 'a -> bool
```


Fortsetzung des Beispiels

```
- val ziffer_menge : int menge =
      fn 0 => true
       | 1 => true
       | 2 => true
       | 3 => true
       | 4 => true
       | 5 => true
       | 6 => true
       | 7 => true
       | 8 => true
       | 9 => true
       | _ => false;
val ziffer_menge = fn : int menge

- ziffer_menge(4);
val it = true : bool

- ziffer_menge(~12);
val it = false : bool
```

Die Sicht einer Menge als Funktion mit der Menge der Boole´schen Werte als Wertebereich ist in der Mathematik und in der Informatik geläufig. Solche Funktionen werden auch „*Charakteristische Funktionen*“ von Mengen genannt.

Überblick

1. Typen im Überblick
2. Deklarationen von Typabkürzungen in SML: type-Deklarationen
3. Definition von Typen: datatype-Deklarationen
4. Definition von rekursiven Typen
5. Beweisprinzip der strukturellen Induktion
6. Beispiele: Implementierungen des Konzepts der „Menge“
7. Beispiele: Grundlegende Funktionen für binäre (Such-) Bäume

datatype- und abstype - Deklarationen

- Eine Typabkürzung definiert keinen neuen Typ.
- Neue Typen können in SML mit datatype- und abstype-Deklarationen vereinbart werden.
- Die Definition von sogenannten „*abstrakten Typen*“ in SML mit abstype-Deklarationen wird später eingeführt.
- In diesem Kapitel werden nur die Definitionen von neuen Typen mit datatype-Deklarationen behandelt.

Definition von Typen mit endlich vielen atomaren Werten

- Ein Typ „Farbe“ bestehend aus der Wertemenge {`Rot,Gelb,Blau`} kann in SML so definiert werden:

```
- datatype Farbe = Rot | Gelb | Blau;
datatype Farbe = Blau | Gelb | Rot

- Rot;
val it = Rot : Farbe
```

- Diese Deklaration legt das Folgende fest:
  - Der Name `Farbe` ist eine Typkonstante.
  - Die Typkonstante `Farbe` wird an die Wertemenge {`Rot,Gelb,Blau`} gebunden.
  - Die Namen `Rot`, `Gelb` und `Blau` sind `0`-stellige (Wert-)Konstruktoren.

datatype – Deklarationen

- Typen, die mit *datatype-Deklarationen* definiert werden, sind neue Typen ohne jegliche Entsprechung in den vordefinierten Typen.
- Ihre (Wert-) Konstruktoren können genauso verwendet werden (u.a. für Pattern Matching) wie die (Wert-) Konstruktoren von vordefinierten Typen.

Beispiel

```
- fun farbname(Rot) = "rot"
    |  farbname(Gelb) = "gelb"
    |  farbname(Blau) = "blau";
val farbname = fn : Farbe -> string

- farbname(Gelb);
val it = "gelb" : string

- [Rot, Blau];
val it = [Rot,Blau] : Farbe list
```

- Die Definition der Funktion `farbname` besitzt einen Fall für jeden (Wert-) Konstruktor des Typs `Farbe`.
- Um Fehler zu vermeiden und für eine gute Lesbarkeit des Programms sollten diese Fälle in der Reihenfolge der Typdefinition aufgelistet werden.

Boole´sche Werte

- Man kann den vordefinierten Typ „Boole´sche Werte“ als einen Typ ansehen, der so hätte definiert werden können:

  ```
  - datatype bool = true | false;
  ```

- Diese Typdeklaration würde die vordefinierten Funktionen des vordefinierten Typs `bool` „ausschalten“:

  Die Bindung der Typkonstanten `bool` an den *benutzerdefinierten* Typ überschattet die alte Bindung derselben Typkonstante an den *vordefinierten* Typ „Boole´sche Werte“.

Die Gleichheit

- In SML verfügen benutzerdefinierte Typen, die Mengen von atomaren Werten bezeichnen, stets über die Gleichheit, die implizit bei der datatype-Deklaration mit definiert wird.
- Beispiel:

```
- Blau = Blau;
val it = true : bool

- Blau = Rot;
val it = false : bool
```


Die Ordnung

- In SML verfügen benutzerdefinierte Typen, die Mengen von atomaren Werten bezeichnen, NICHT über eine implizit definierte Ordnung.
- Beispiel:

```
- Rot < Gelb;
Error: overloaded variable not defined at type
       symbol: <
       type: Farbe
```

Austauschbarkeit von Typdeklarationen in SML

- Die folgenden Typdeklarationen sind in SML austauschbar:

```
- datatype Farbe = Rot | Gelb | Blau;
- datatype Farbe = Gelb | Rot | Blau;
```

- Andere Programmiersprachen würden aus der Reihenfolge der (Wert-) Konstruktoren in der Typdefinition die Ordnung

  `Rot < Gelb < Blau`

  implizit definieren.

Definition von Typen mit zusammengesetzten Werten

- Typen mit zusammengesetzten Werten können auch definiert werden:

```
- datatype Preis = DM of real | EURO of real;
datatype Preis = DM of real | EURO of real
```

- Diese Deklaration legt das Folgende fest:
  - Der Name `Preis` ist eine Typkonstante.
  - Die Typkonstante `Preis` wird an die Wertemenge (in mathematischer Notation) $\{DM(x) \mid x \in real\} \cup \{EURO(x) \mid x \in real\}$ gebunden.
  - Die Namen `DM` und `EURO` sind unäre (einstellige) (Wert-)Konstruktoren, beide vom Typ: `real -> Preis`

Beispiel

```
- DM(4.5);
val it = DM 4.5 : Preis

- EURO(2.0);
val it = EURO 2.0 : Preis

- DM(1);
Error: operator and operand don't agree [literal]
  operator domain: real
  operand:         int
  in expression:
     DM 1
```


Definition mit Pattern Matching

- Die bevorzugte Weise, Funktionen auf benutzerdefinierten Typen mit zusammengesetzten Werten zu definieren, ist das Pattern Matching.
- Beispiel:

```
- fun wechseln( DM(x) ) = EURO(0.51129 * x)
    |  wechseln( EURO(x) ) = DM(1.95583 * x);
val wechseln = fn : Preis -> Preis
```

Gleichheit für Typen mit zusammengesetzten Werten

- Benutzerdefinierte Typen, die Mengen von zusammengesetzten Werten bezeichnen, verfügen in SML über die Gleichheit.
- Sie ist komponentenweise definiert:

```
- datatype zeitpunkt = Sek of int | Min of int | Std of int;
datatype zeitpunkt = Min of int | Sek of int | Std of int

- Sek(30) = Sek(0030);
val it = true : bool

- Min(60) = Std(1);
val it = false : bool

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


Definition von Gleichheit

- Zwei Werte sind gleich, wenn:
  - ihre (Wert-) Konstruktoren gleich sind
    (was für `Min(60)` und `Std(1)` nicht der Fall ist)

  und
  - diese (Wert-) Konstruktoren auf Vektoren angewandt werden, die ebenfalls gleich sind.

    Dabei muss berücksichtigt werden, dass in SML ein einstelliger Vektor mit seiner einzigen Komponente gleich ist.

Keine vordefinierte Gleichheit

- Ist die Gleichheit auf dem Typ einer Komponente NICHT definiert, so ist sie es auch nicht auf dem benutzerdefinierten Typ:

```
- DM(2.0) = DM(2.0);
Error: operator and operand don't agree [equality
type required]
    operator domain:  ''Z * ''Z
    operand:          Preis * Preis
    in expression:
       DM 2.0 = DM 2.0
```

- Erinnerung: In SML ist die Gleichheit über den Gleitkommazahlen nicht vordefiniert (Kapitel 5.2).

„Typenmix“

- Typdeklarationen ermöglichen die Definition von Funktionen mit Parametern unterschiedlicher Grundtypen:

```
- datatype int_or_real = Int of int | Real of real;
datatype int_or_real = Int of int | Real of real

- fun round(Int(x)) = Int(x)
    | round(Real(x)) = Int(trunc(x));
val round = fn : int_or_real -> int_or_real

- round(Int(56));
val it = Int 56 : int_or_real

- round(Real(56.8976));
val it = Int 56 : int_or_real
```

Beispiel zur Verwendung des Typs `int_or_real`

```
- datatype int_or_real = Int of int | Real of real;
- local
    fun real_abl f x = let val delta = 1E~10
                       in  (f(x + delta) - f(x)) / delta
                       end;
    fun konvertieren(Int(x)) = real(x)
      | konvertieren(Real(x)) = x
  in
    fun abl f (Int(x)) = Real(real_abl f (real(x)))
      | abl f (Real(x)) = Real(real_abl f x)
  end;
val abl = fn : (real -> real) -> int_or_real -> int_or_real
- abl (fn x => 2.0 * x) (Int(5));
val it = Real 2.00000016548 : int_or_real
- abl (fn x => 2.0 * x) (Real(5.0));
val it = Real 2.00000016548 : int_or_real
```

Überblick

1. Typen im Überblick
2. Deklarationen von Typabkürzungen in SML: type-Deklarationen
3. Definition von Typen: datatype-Deklarationen
4. Definition von rekursiven Typen
5. Beweisprinzip der strukturellen Induktion
6. Beispiele: Implementierungen des Konzepts der „Menge“
7. Beispiele: Grundlegende Funktionen für binäre (Such-) Bäume

Werte unbegrenzter Größe

- Ein Typ kann Werte unbegrenzter (aber endlicher) Größe haben.
- Beispiele:
  - Listen
  - Binärbäume

Wurzel, Blätter, Äste, Bäume und Wälder

Definition ((gerichteter) Graph)
Ein (gerichteter) Graph G ist ein Paar (Kn,Ka) mit $Ka \subseteq Kn \times Kn$.
Die Elemente von Kn werden *Knoten* von G genannt.
Die Elemente von Ka heißen die *Kanten* von G.
Ist Kn leer, so sagt man, dass der Graph $G = (Kn,Ka) = (\{\},\{\})$ leer ist.
Ist $(k_1, k_2) \in Ka$, so heißt k_2 ein Nachfolger von k_1 (in G).

Definition (zusammenhängender Graph)

Definition (zusammenhängender Graph)
Ein (gerichteter) Graph $G = (Kn,Ka)$ heißt zusammenhängend, wenn es für jedes Paar (k_a, k_e) von Knoten eine Folge (*Pfad* genannt) von Knoten $k_1, \ldots, k_m \in Kn$ $(m \geq 1)$ gibt, so dass:

- $k_1 = k_a$
- $k_m = k_e$
- für jedes $i \in \mathbf{N}$ mit $1 \leq i \leq m$ -1 ist $(k_i, k_{i+1}) \in Ka$ oder $(k_{i+1}, k_i) \in Ka$.

- D.h. man kann in einem zusammenhängenden Graph G von jedem Knoten von G über die Kanten von G jeden anderen Knoten von G erreichen, wobei die Kanten in beliebiger Richtung durchlaufen werden dürfen.

Definition (Zyklus in einem Graph)

Definition (Zyklus in einem Graph)
Ein Zyklus in einem Graph $G = (Kn,Ka)$ ist eine endliche Folge $k_1, \ldots, k_m$ $(m \geq 1)$ von Knoten, so dass:

- für jedes $i \in \mathbf{N}$ mit $1 \leq i \leq m$ -1 ist $(k_i, k_{i+1}) \in Ka$
- $(k_m, k_1) \in Ka$.

- D.h. ein Zyklus ist ein Rundgang durch den Graph über die Kanten des Graphen, bei dem die Richtung der Kanten eingehalten wird.

Definition (Baum, Wurzel, Blatt)

Definition (Baum, Wurzel, Blatt)
Sei K eine Menge (K ist eine Referenzmenge für die Knoten. D.h. dass alle Knoten der Bäume, die im Folgenden definiert werden, Elemente von K sind). *Bäume* (mit Knoten in K) werden wie folgt definiert:

1. Der leere Graph $(\{\},\{\})$ ist ein Baum (mit Knoten in K). Dieser Baum hat keine *Wurzel*.
2. Für jedes $k \in K$ ist $(\{k\},\{\})$ ein Baum (mit Knoten in K). Die *Wurzel* dieses Baums ist der Knoten k.
3. Ist $m \in \mathbf{N}\backslash\{0\}$ und ist (Kn_i,Ka_i) für jedes $i \in \mathbf{N}$ mit $1 \leq i \leq m$ ein Baum mit Wurzel k_i , so dass die Knotenmengen Kn_i paarweise disjunkt sind, und ist $k \in K$ ein „neuer" Knoten, der in keinem Kn_i vorkommt, so ist (Kn,Ka) ein Baum (mit Knoten in K) wobei

   $$Kn = \{k\} \cup \bigcup_{i=1}^{m} Kn_i$$

   $$Ka = \{(k, k_i) | \ 1 \leq i \leq m \ \} \cup \bigcup_{i=1}^{m} Ka_i$$

   Die *Wurzel* dieses Baums ist der Knoten k.

Die Knoten eines Baums, die keine Nachfolger haben, heißen *Blätter*.

Definition (Binärbaum)

Definition (Binärbaum)
Ein Binärbaum B ist ein Baum mit der Eigenschaft:
Jeder Knoten von B ist entweder ein Blatt oder hat genau zwei Nachfolger.

Zusammenfassung: Bäume

- Es ist gut zu wissen, dass:
  - jeder nichtleere Baum eine Wurzel hat, und dass aus der Wurzel jeder andere Knoten des Baumes über die Kanten des Baumes erreicht werden kann, wobei die Richtung der Kanten eingehalten wird
  - in einem Baum kein Zyklus vorkommt
  - jeder Baum einen Pfad von seiner Wurzel zu jedem seiner Blätter enthält (Ein Wurzel-Blatt-Pfad in einem Baum heißt *Ast*).

Definition (Wald)

Definition (Wald)
Ein Wald ist eine Menge von Bäumen, deren Knotenmengen paarweise disjunkt sind.

- Bäume und Wälder werden in der Informatik häufig verwendet.
- Fast alle Bäume werden graphisch so dargestellt, dass ihre Wurzel oben und ihre Blätter unten sind.

Induktive Definition

- Eine Definition wie die Definition der Bäume heißt „*induktive Definition*“.
- Induktive Definitionen bestehen immer aus einem Basisfall oder mehreren Basisfällen (Fall 2) und einem Induktionsfall oder mehreren Induktionsfällen (Fall 3).
- Zudem können sie Sonderfälle (Fall 1) besitzen.
- Die Basisfälle bestimmen Anfangsstrukturen.
- Die Induktionsfälle sind Aufbauregeln.

Induktive Definition und rekursive Algorithmen

- Ist eine Datenstruktur (wie die Datenstruktur „Baum“) induktiv definiert, so lassen sich Algorithmen über dieser Datenstruktur leicht rekursiv spezifizieren.
- Die Spezifikation von rekursiven Algorithmen über induktiv definierten Datenstrukturen ist eine grundlegende Technik der Informatik.

Rekursive Ermittlung der Anzahl der Knoten eines Baumes

- Die Anzahl der Knoten eines Baumes kann man leicht rekursiv ermitteln:
  1. Der leere Baum hat 0 Knoten.
  2. Ein Baum der Gestalt $(\{k\}, \{\})$ hat 1 Knoten.
  3. Ein Baum der Gestalt (Kn,Ka) mit

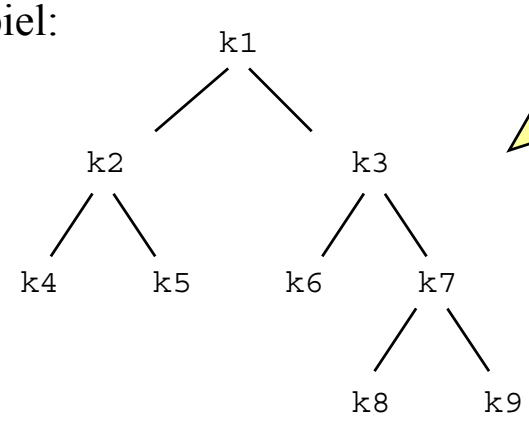
$$Kn = \{k\} \cup \bigcup_{i=1}^{m} Kn_i$$

$$Ka = \{(k, k_i) | \ 1 \le i \le m \ \} \cup \bigcup_{i=1}^{m} Ka_i$$

hat einen Knoten mehr als die Summe der Knotenanzahlen der Bäume (Kn_i,Ka_i) mit $1 \le i \le m$.

Darstellung von Bäumen: graphisch und durch Ausdrücke

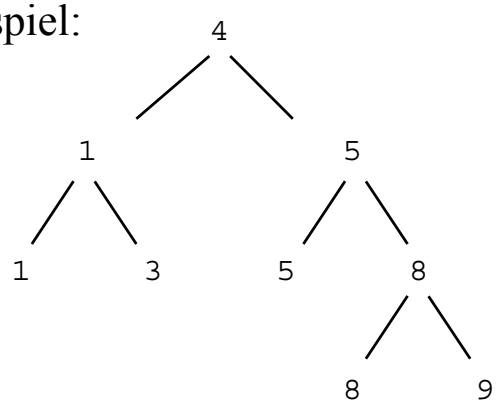
- Man kann Bäume graphisch veranschaulichen:
  - Knoten werden durch Symbole dargestellt
  - Kanten sind Verbindungslinien dazwischen (wobei die Richtung der Kanten fast immer mit der Richtung von oben nach unten gleichgesetzt wird)
- Beispiel:



Dieser Baum ist ein Binärbaum mit Wurzel `k1` und Blättern `k4`, `k5`, `k6`, `k8`, `k9`. In vielen Fällen sind die Bezeichner der Knoten irrelevant, so dass man an Stelle von `k1`, `k2` usw. jeweils einfach einen kleinen Kreis zeichnet oder ein Symbol wie `*` .

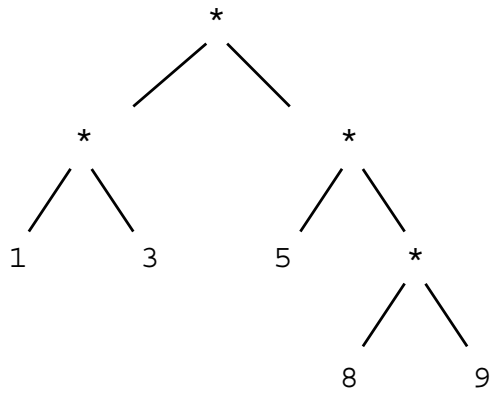
Darstellung von Bäumen: graphisch mit Werten

- Oft werden die Knoten zusätzlich mit Werten markiert, z.B. mit Zahlen.
- Die Werte können mehrfach im Baum vorkommen, während die Knoten selbst nicht mehrfach vorkommen können.
- Man zeichnet die Markierungen statt der Knoten oder `*`.
- Beispiel:



Darstellung von Bäumen: graphisch mit bewerteten Blättern

- In der Informatik werden auch nur die Blätter mit Werten markiert, so dass sich eine Mischform zur Darstellung ergibt:
- Beispiel:



Darstellung von Bäumen: durch Ausdrücke (1)

- Eine andere Darstellung beruht auf Ausdrücken, wobei die Kanten des Baums durch die Verschachtelung von Teilausdrücken dargestellt werden.
- Der Baum ohne Markierungen von Knoten kann so dargestellt werden:

```
Knt    (Knt( Blt ,
             Blt),
        Knt( Blt ,
             Knt( Blt,
                  Blt)))
```

- Dieser Ausdruck ist mit zwei verschiedenen Symbolen gebildet, mit denen Blätter und andere Knoten unterschieden werden.

Darstellung von Bäumen: durch Ausdrücke (2)

- Es ist (nur) eine Konvention, dass die Reihenfolge der Argumente in den Ausdrücken der Reihenfolge der Nachfolger in der graphischen Darstellung entspricht.
- Da man die Struktur von verschachtelten Ausdrücken durch Einrückungen verdeutlicht, wirkt die Darstellung durch Ausdrücke wie eine Spiegelung aus der graphischen.
- Wenn man die Darstellung durch Ausdrücke für die Varianten mit Markierungen betrachtet, wird das deutlicher.

Darstellung von Bäumen: durch Ausdrücke (3)

- Für die Varianten mit Markierungen sind zusätzliche Argumente für die Markierungen erforderlich:

```
Knt (4,
     Knt ( 1,
          Blt(1),
          Blt(3)),
     Knt ( 5,
          Blt (5),
          Knt (8,
               Blt(8),
               Blt(9))))
```

- Hier kann man ebenso wie in der graphischen Darstellung auch Varianten betrachten, in denen nur die Blätter mit Werten markiert sind.

Rekursive Typen zum Ausdrücken von induktiven Definitionen

- In SML können nichtleere Binärbäume, deren Blätter mit ganzen Zahlen markiert sind, so spezifiziert werden:

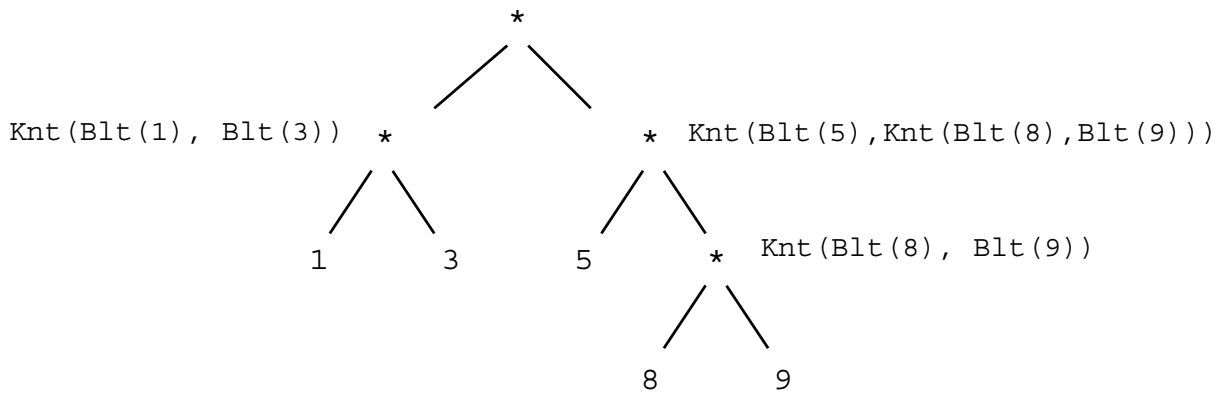
`Blt` für Blatt
`Knt` für Knoten

```
- datatype binbaum1 = Blt of int
                    | Knt of binbaum1 * binbaum1;
datatype binbaum1 = Blt of int
                  | Knt of binbaum1 * binbaum1

- val b = Knt(Knt(Blt(1),Blt(3)),
              Knt(Blt(5),Knt(Blt(8),Blt(9))));
val b = Knt( Knt(Blt #,Blt #),
             Knt (Blt #,Knt #)) : binbaum1
```

- Der (Wert-) Konstruktor `Knt` bildet aus zwei Binärbäumen einen neuen Binärbaum.
- Sein Typ ist `binbaum1 * binbaum1 -> binbaum1`.

Graphische Darstellung

- Der Wert von `b` wird abgekürzt gedruckt.
- Die Abkürzung betrifft nur die interaktive Treiberschleife von SML – in der globalen Umgebung ist der richtige Wert abgespeichert.
- Man kann diesen Binärbaum graphisch so darstellen:



Beblätterter Binärbaum

- Ein solcher Binärbaum wird „*beblättert*“ genannt, weil seine Blätter (und nur seine Blätter) Markierungen (Werte) tragen.
- Später wird eine andere Art von Binärbäumen eingeführt – der Binärbaum mit Knotenmarkierungen – deren Knoten alle Markierungen (Werte) tragen.

Rekursive Typdefinitionen und Typen

- Die vorherige Typdefinition der Binärbäume hat dieselbe Struktur wie die Definition der Bäume am Anfang des Abschnitts:
  - Inhaltlich unterscheidet sie sich nur in der zusätzlichen Einschränkung, dass jeder Knoten, der kein Blatt ist, genau zwei Nachfolger haben muss.
  - Syntaktisch unterscheidet sie sich in der Verwendung von SML-Konstrukten.
- Eine Typdefinition wie die Definition des Typs `binbaum1` ist eine *induktive* Definition.
- Wegen ihrer syntaktischen Ähnlichkeit mit rekursiven Algorithmen werden solche Typdefinitionen in der Informatik manchmal „*rekursive Typdefinitionen*“ genannt.
- Die Typen, die mittels induktiven (oder rekursiven) Typdefinitionen vereinbart werden, heißen „*rekursive Typen*“.

`blaetterzahl`

- Die Funktion `blaetterzahl` zur Berechnung der Anzahl der Blätter eines Binärbaums vom Typ `binbaum1` kann in SML so implementiert werden:

```
- fun blaetterzahl (Blt(_)) = 1
  | blaetterzahl (Knt(b1, b2)) =  blaetterzahl b1 +
                                  blaetterzahl b2;
val blaetterzahl = fn : binbaum1 -> int

- val c = Knt(Blt(1), Blt(2));

- blaetterzahl(c);
val it = 2 : int

- blaetterzahl(Knt(c, Knt(c, c)));
val it = 6 : int
```

Polymorphe Typen

- Oft ist es vorteilhaft, rekursive Typen polymorph zu definieren.
- Ein polymorpher Typ `binbaum1` kann so definiert werden:

```
- datatype 'a binbaum2 = Blt of 'a
                       | Knt of 'a binbaum2 * 'a binbaum2;
datatype 'a binbaum2 = Blt of 'a
                     | Knt of 'a binbaum2 * 'a binbaum2
```


`blaetterzahl` als polymorphe Funktion (1)

```
- fun blaetterzahl (Blt(_)) = 1
    | blaetterzahl (Knt(b1, b2)) =  blaetterzahl b1 +
                                    blaetterzahl b2;
val blaetterzahl = fn : 'a binbaum2 -> int

- val d = Knt(Blt("ab"), Blt("cd"));
val d = Knt (Blt "ab",Blt "cd") : string binbaum2

- blaetterzahl(d);
val it = 2 : int

- let val e = Knt(d, d);
      val f = Knt(e, e)
  in
      blaetterzahl(Knt(f, f))
  end;
val it = 16 : int
```

blaetterzahl als polymorphe Funktion (2)

```
- val g = Knt(Blt([1,2,3]), Blt[4,5]);
val g = Knt (Blt [1,2,3],Blt [4,5]) : int list binbaum2

- let val h = Knt(g, g);
      val i = Knt(h, h)
  in
      blaetterzahl(Knt(i, i))
  end;
val it = 16 : int
```

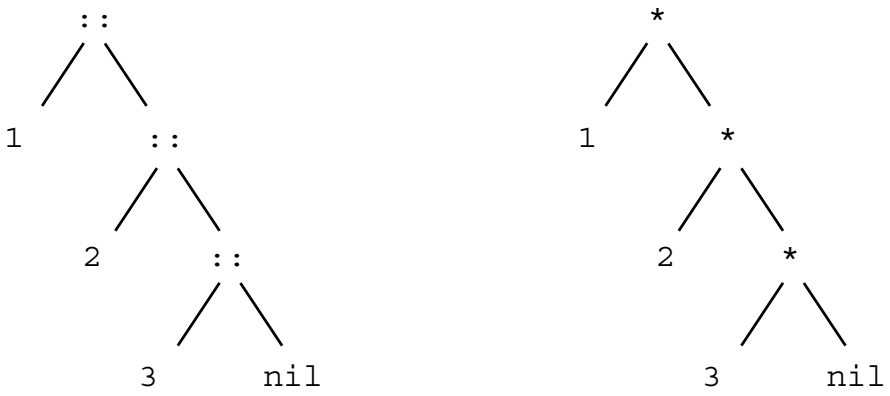

Suche in Binärbäumen

- Mit dem folgenden Prädikat kann überprüft werden, ob ein Element in einem beblätterten Binärbaum vorkommt:

```
- fun suche(x, Blt(M)) = (x = M)
    | suche(x, Knt(B1, B2)) = suche(x, B1)
                              orelse suche(x, B2);
val suche = fn : 'a * binbaum2 -> bool

- val b = Knt(Knt(Blt(1), Blt(3)), Knt(Blt(5),
              Knt(Blt(8), Blt(9))));
val b = Knt(Knt(Blt #,Blt #),Knt(Blt #,Knt #)) : int binbaum2

- suche(5, b);
val it = true : bool

- suche(12, b);
val it = false : bool
```

Die Liste als beblätterter Binärbaum

- Die Liste ist ein Sonderfall des Binärbaums:



- Die Suche in so einem beblätterten Binärbaum besteht aus den selben Schritten wie die Suche in einer Liste mit der Funktion `member`.

Überblick

1. Typen im Überblick
2. Deklarationen von Typabkürzungen in SML: type-Deklarationen
3. Definition von Typen: datatype-Deklarationen
4. Definition von rekursiven Typen
5. Beweisprinzip der strukturellen Induktion
6. Beispiele: Implementierungen des Konzepts der „Menge“
7. Beispiele: Grundlegende Funktionen für binäre (Such-) Bäume

Strukturelle Induktion

- Wie können Eigenschaften von Funktionen bewiesen werden, die auf rekursiven Typen definiert sind?
- Induktionsbeweise und induktive Definitionen haben eine ähnliche Gestalt :
  - Induktionsbeweise und induktive Definitionen weisen Basisfälle und Induktionsfälle auf.
  - Sowohl bei Induktionsbeweisen als auch bei induktiven Definitionen stellen die Induktionsfälle eine Art stufenweisen Aufbau dar.
- Eigenschaften von Funktionen, die auf rekursiven Typen definiert sind, lassen sich oft induktiv beweisen.
- Das Beweisprinzip der vollständigen Induktion bezieht sich auf natürliche Zahlen.
- Das Beweisprinzip, das hier anzuwenden ist, heißt „*strukturelle Induktion*“

Beweis

- Sei `t` ein rekursiver Typ mit den (Wert-) Konstruktoren $k_i^{s_i}$ für $0 \le i \le I$, wobei s_i die Stelligkeit des (Wert-) Konstruktors $k_i^{s_i}$ ist.
- Um zu zeigen, dass alle Werte des Typs `t` eine Eigenschaft **E** (wie etwa: “die Auswertung einer Anwendung der Funktion *f* auf den Wert terminiert“) besitzen, genügt es zu zeigen:
  1. Basisfälle: Jeder 0-stellige (Wert-) Konstruktor k_i^0 $(0 \le i \le I)$ besitzt die Eigenschaft **E**.
  2. Induktionsfälle:
     Für jeden (Wert-) Konstruktor $k_i^{s_i}$ (der Stelligkeit s_i) mit $i \ge 1$ gilt:
     immer wenn Werte $W_1, \ldots, W_{si}$ des Typs `t` die Eigenschaft **E** besitzen (Induktionsannahme), dann besitzt auch der Wert $k_i^{s_i}(W_1, \ldots, W_{si})$ des Typs `t` die Eigenschaft **E**.

qed.

Das Beweisprinzip der strukturellen Induktion lässt sich auf die vollständige Induktion zurückführen (das ist aber nicht unmittelbar – siehe die Hauptstudiums-Vorlesung „Logik für Informatiker“).

Beispiel zur Anwendung der strukturellen Induktion

- Als Beispiel einer Anwendung des Beweisprinzips der strukturellen Induktion kann man zeigen, dass jede Anwendung der polymorphen Funktion `blaetterzahl` auf einen Binärbaum vom Typ `binbaum1` terminiert.

Beweis

- Basisfall:
  Ist `A` ein Ausdruck der Gestalt `Blt(x)`, so führt die Auswertung von `blaetterzahl(A)` nach Definition (der Funktion `blaetterzahl`) zur Auswertung von `1`, was offenbar terminiert.
- Induktionsfall:
  Seien `W1` und `W2` zwei Werte vom Typ `binbaum1`.
- Induktionsannahme:
  Sei angenommen, dass die Auswertungen von `blaetterzahl(W1)` und von `blaetterzahl(W2)` beide terminieren.
- Nach Definition der Funktion `blaetterzahl` führt die Auswertung von `blaetterzahl(Knt(W1,W2))` zur Auswertung von `blaetterzahl(W1)+ blaetterzahl(W2)`.
  Nach Induktionsannahme terminieren die Auswertungen der beiden Komponenten dieser Addition. Folglich terminiert auch die Auswertung dieses Ausdrucks.

qed.