

10. Konstruktortypen (konkrete Datentypen)

10.1 Typdefinition

type t = <Typausdruck>

führt t als Abkürzung für den durch den Typausdruck ein.

Es werden keine **neuen** Typen definiert.

z.B. type inttelnr = {...}

Der Typausdruck kann Variablen enthalten.

Beispiel: Implementierung von endlichen Abbildungen als Mengen von Paaren

type ('a,'b) abb = ('a * 'b) list

val wertung : abb = [("x", (val,1)), ("y", (val,3))]

10.2 Datentyp-Deklarationen

Einführung **neuer**, eventuell **rekursiver** Datentypen

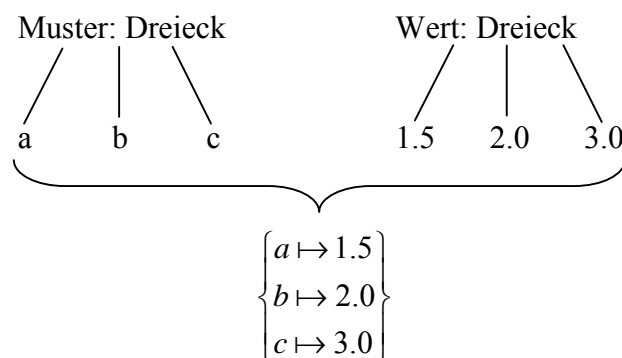
Datentyp-Deklarationen führen ein:

- ein oder mehrere **Typ-Konstruktoren**, eventuell simultan rekursiv;
0-stellige Typ-Konstruktoren sind **Typen**
- ein oder mehrere **Wert-Konstruktoren** für jeden eingeführten Typkonstruktor;
0-stellige Wert-Konstruktoren sind **Werte**.

Beispiele: <http://rw4.cs.uni-sb.de/~joba/Info1/Material/datatypes.sml>

Musterabgleich:

Muster und Werte mit Wertkonstruktoren



10.3 Aufzählungstypen (enumeration types)

definiert durch Menge von **0-stelligen** Wertkonstruktoren

Vordefinierte Aufzählungstypendatatype bool = true | falsedatatype order = LESS | EQUAL | GREATER

Eingebaute Vergleichsfunktionen mit order als Ergebnistyp:

Int.compare, Real.compare, Char.compare, String.compare, Time.compare, Date.compare

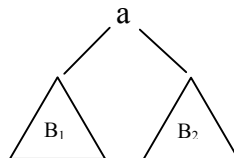
$$\text{String.compare}(x, y) = \begin{cases} \text{LESS falls } x <_{\text{lex}} y \\ \text{EQUAL falls } x = y \\ \text{GREATER falls } y <_{\text{lex}} x \end{cases}$$

<_{lex} lexikographische Ordnung**Beispiel:**

Funktion, die eine Liste von ganzen Zahlen bekommt und die Zahlen <0, =0, >0 in der Liste zählt

10.4 Rekursive Datentypen

Bisherige rekursive Datentypen: Listen-Typen

Mit datatype-Deklarationen kann man neue, rekursive Datentypen deklarieren.**Beispiel:** knotenmarkierte binäre Bäume**Induktive Definition**Basis: Der leere Baum ist ein binärer BaumInd.S: Sind B₁ und B₂ binäre Bäume und ist a eine Knotenmarkierung, so a ein binärer Baum

zuerst: binärer Baum markiert mit ganzen Zahlen

datatype ibintree

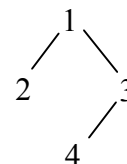
Empty |

Node of int * ibintree * ibintreeNeuer Typkonstruktor: ibintree (nullstellig)Neue Wertkonstruktoren: Empty (0-stellig), Node (3-stellig)

```

val b = Node (1,
  Node (2, Empty, Empty),
  Node (3,
    Node (4, Empty, Empty),
    Empty))

```



Baumstruktur kann unabhängig vom Markierungsalphabet definiert werden, Markierungsalphabet kann “herausabstrahiert” werden.

Beispiele: <http://rw4.cs.uni-sb.de/~joba/Info1/Material/datatypes.sml>

Rechnen auf rekursiven Datentypen

Höhe eines Baums: height = Länge des maximalen Pfades

Größe eines Baums: size = Zahl der Knoten

Bäume mit beliebiger Stelligkeit

Kinder eines Knotens als Liste von Bäumen

Alternative: Wälder

datatype 'a tree = Empty |

Node of 'a * 'a forest

and 'a forest = Nil |

Cons of 'a tree * 'a forest

10.5 Der option-Datentyp

Eine Funktion $f:A \rightarrow B$ heißt **partiell**, wenn sie nicht für alle $a \in A$ definiert ist.

Beispiele: Fakultätsfunktion, definiert auf \mathbb{Z}
hd, tl für Listen

Wie drückt sich partielle Definiertheit beim Programmieren von Funktionen in SML aus?

1. Anwendung der Funktion auf ein solches Argument terminiert nicht, z.B. die früher definierte Funktion fact angewendet auf -1
2. Anwendung terminiert durch Auslösen einer Ausnahme (exception)
3. Anwendung der Funktion ergibt ein spezielles Ergebnis mit der Interpretation: Funktion ist auf Argument undefiniert.

Letzteres lässt sich durch den vordefinierten Datentyp 'a option ausdrücken:

datatype 'a option = NONE | SOME of 'a

Eine partiell definierte Funktion gibt NONE zurück für Argumente, auf denen sie nicht definiert ist, und SOME v für Argumente, auf denen sie mit Ergebnis v definiert ist.

Die Funktion valOf: 'a option \rightarrow 'a entfernt das SOME von SOME v

Beispiel:

Fakultätsfunktion, welche auf $n < 0$ testet und in diesem Fall den Wert NONE zurückgibt:

fun neuFact n = if n < 0 then NONE else SOME (fact n)

oder

```

fun neufact n =
  case Int.compare (n,0) of
    GREATER => SOME (n * valOf (neufact (n-1)))
  | EQUAL => SOME (1)
  | LESS => NONE

```

Weitere Verwendung des option-Datentyps:

- Optionale Argumente
Default angenommen, wenn Argument fehlt
- Optionale Komponenten in zusammengesetzten Datentypen

```

type inttelnr = { land:int , amt:int option , anschl:int }
val lichttelnr = { land=4175 , amt=NONE , anschl=... }
val praes = { land=49 , amt=SOME (681) , anschl=3021000 }

```

10.6 Binäre Suchbäume (binary search trees, BST)

Datenstruktur zum effizienten Suchen von Daten aus einem geordneten Bereich

10.6.1 Die BST-Eigenschaft

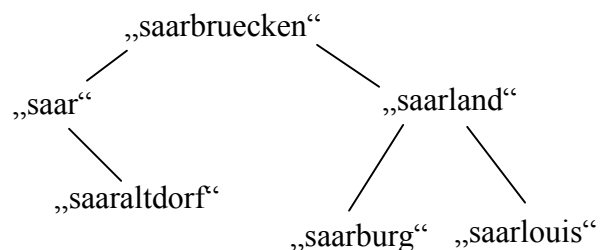
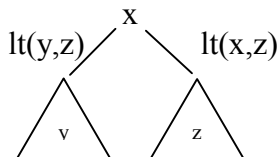
Abgespeicherte Daten sind die Markierungen der Knoten. Sie stammen aus einem Bereich mit einer Relation lt mit den Eigenschaften

1. Irreflexivität: $\forall x \in D$: nicht $lt(x,x)$
2. Transitivität: $\forall x, y, z \in D$: Aus $lt(x,y)$ und $lt(y,z)$ folgt $lt(x,z)$
3. Vergleichbarkeit: $\forall x, y$: Wenn $x \neq y$, dann entweder $lt(x,y)$ oder $lt(y,x)$

Beispiele für lt : $<$ auf int , $char$, $string$,...

Elemente aus D markieren die Knoten der Binärbäume so, dass (BST) Für jeden inneren Knoten n markiert mit x gilt:

- für alle Markierungen y im linken Unterbaum von n gilt: $lt(y,x)$
- für alle Markierungen z im rechten Unterbaum von n gilt: $lt(x,z)$



10.6.2 Nachschlagen in binären Suchbäumen

Aufgabe: Stelle fest, ob ein gegebenes Element $x \in D$ im binären Suchbaum vorkommt

Schlüssel für die Effizienz: nur ein Pfad muss durchsucht werden!

Begründung: Aus der BST-Eigenschaft folgt:

Nach Vergleich von x mit Markierung y eines inneren Knotens kann maximal ein Unterbaum zur Suche übrigbleiben:

- bei $lt(x,y)$ der linke Unterbaum
- bei $lt(y,x)$ der rechte Unterbaum
- bei $x = y$ Ende der Suche

Beispiele: <http://rw4.cs.uni-sb.de/~joba/Info1/Material/bst.sml>

10.6.3 Einfügen in einen binären Suchbaum

Aufgabe: Einfügen eines gegebenen Elements x unter Erhaltung der BST-Eigenschaft

Ergebnis: ein eventuell modifizierter Baum

Beispiele: <http://rw4.cs.uni-sb.de/~joba/Info1/Material/bst.sml>

10.6.4 Laufzeit

Analyse von lookup: Laufzeit als Zahl der rekursiven Funktionsanwendungen

Fall Empty: 0

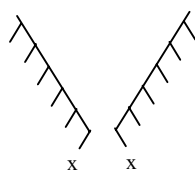
Fall Node ($y, \text{left}, \text{right}$):
 1 + Laufzeit für lookup ($\dots, \text{left}, \dots$)
 1 + Laufzeit für lookup ($\dots, \text{right}, \dots$)
 0 falls $x = y$

In den rekursiven Fällen ist der zu durchsuchende Pfad um 1 kürzer geworden.

Also: Zahl der Funktionsanwendungen = $c \cdot$ Länge des Pfades von der Wurzel bis zu dem mit x markierten Knoten oder einem Blatt, falls x nicht vorkommt.

Länge dieser Pfade (als Funktion der Elemente n im Baum):

1. Günstigster Fall: x ist Wurzelmarkierung: 0 Anwendungen
2. Ungünstigster Fall: n



degenerierte Bäume, entstehen bei Einfügen von sortierten Folgen von Daten durch insert

Wie kann man die **Laufzeit im schlechtesten Fall** begrenzen?

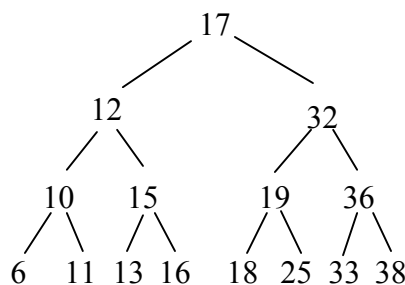
Erfahrung: degenerierte Bäume liefern den schlechtesten Fall.

1. Kann man verhindern, dass Bäume degenieren?
2. Wenn ja, wie schlecht ist dann der schlechteste Fall?

Definition (vollständiger binärer Baum)

Ein binärer Baum heißt vollständig, wenn für jeden inneren Knoten gilt: linker und rechter Unterbaum sind gleich tief.

Beispiel: vollständiger binärer Suchbaum



Höhe: Länge des maximal langen Pfades: 3

Größe: Zahl der Knoten = $2^{3+1} - 1$
 $= 2^{\text{Höhe}+1} - 1$

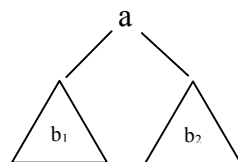
Satz: Die Größe eines nichtleeren vollständigen binären Baums der Höhe k ist $2^{k+1} - 1$

Beweis: Induktion über k

Ind.Anf.: $k=0$, Größe $1 = 2^{0+1} - 1$

Ind.Schritt: $k \rightarrow k+1$

Für zwei Bäume b_1, b_2 der Höhe k gilt nach Induktionsannahme, dass ihre Höhe $2^{k+1} - 1$ ist



vollständiger Baum
der Höhe $k+1$

$$\text{Größe: } 1 + 2^{k+1} - 1 + 2^{k+1} - 1 = 2^{k+2} - 1 \quad \text{q.e.d.}$$

Korollar: Die Höhe eines vollständigen binären Baums der Größe $n > 0$ ist $\log_2(n+1) - 1$.

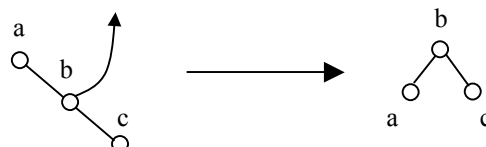
Wenn es gelingt, die binären Suchbäume nahezu vollständig zu halten, dann ist die Laufzeit von lookup logarithmisch.

„nahezu vollständig“ heißt **balanciert**, die Pfade von jedem inneren Knoten zu den erreichbaren Blättern können sich in ihrer Länge um höchstens 1 unterscheiden.

Später lernen wir Datenstrukturen kennen, die immer balanciert sind. Das Mittel dazu sind

Rotationen.

Rotation:



10.7 Abstrakte Syntax

- **konkrete Syntax:** beschreibt den Aufbau von Programmen, wie sie in einen Interpreter oder Übersetzer eingegeben werden (externe Darstellung).
- **abstrakte Syntax:** interne Darstellung von Programmen in Übersetzern oder Interpretern

Definition der abstrakten Syntax als Datentyp erlaubt die Definition eines Interpreters durch eine Reihe von Fällen, welche zu den Wertkonstruktoren korrespondieren.

Beispiele: <http://rw4.cs.uni-sb.de/~joba/Info1/Material/asyntax.sml>

10.8 Syntax, Typregeln, Semantik

Definition und Benutzung von Konstruktortypen

Syntax:

- Deklarationen: $\text{datbind} ::= \text{tycon} = \text{conbind}$
 $\text{conbind} ::= \text{vid} <\text{of ty}> <|\text{conbind}>$
- Muster: $\text{pat} ::= \text{vid atpat} \text{ (vid Wertkonstruktor)}$

Typregeln:

eine neue Relation für Datentypdeklarationen:

$$T_c \subseteq T_U \times \text{tycon} \times \text{conbind} \times T_U$$

Notation: $T_c \subseteq T_U \times \text{tycon} \times \text{conbind} \times T_U$

„In der Typumgebung Γ werden der Typkonstruktor tc und gemäß cb die Wertkonstruktoren von tc gebunden und ergeben die Typumgebung Γ' .“

Wert-Konstruktor-Regeln:

$$\begin{array}{ll}
 [\text{cons2}] & \frac{}{\Gamma, tc \vdash x \supseteq \{x \mapsto (\text{con}, tc)\}} \quad \text{nullstellige Wertkonstruktoren} \\
 [\text{cons3}] & \frac{}{\Gamma, tc \vdash x \text{ of } t \supseteq \{x \mapsto (\text{con}, t \rightarrow tc)\}} \quad \text{nichtnullstellige Wertkonstruktoren} \\
 [\text{datadec}] & \frac{\Gamma, tc \vdash cb \supseteq \Gamma'}{\Gamma \vdash \text{datatype } tc = cb \triangleright \Gamma \oplus \Gamma'} \quad tc \in \text{tycon}
 \end{array}$$

Semantik:

neue Relation $W_c \subseteq \text{conbind} \times W_U$

Notation: $\vdash cb \nearrow \rho$

„Wertkonstruktordefinition cb liefert Wertumgebung ρ “

Wertkonstruktoren:

$$[\text{con2}] \quad \frac{}{\vdash x \nearrow \{x \mapsto (\text{con}, x)\}}$$

$$[\text{con3}] \quad \frac{}{\vdash x \text{ of } t \nearrow \{x \mapsto (\text{con}, x)\}}$$

Musterabgleich:

$$[\text{vid2}] \quad \frac{}{\rho, x \vdash x \nearrow \{}} \quad \rho(x) = (\text{con}, x)$$

$$[\text{vid2F}] \quad \frac{}{\rho, x' \vdash x \nearrow \perp} \quad \begin{array}{l} \rho(x) = (\text{con}, x') \\ x \neq x' \end{array}$$

$$[\text{con}] \quad \frac{\rho, v \vdash p \nearrow \rho'}{\rho, x \text{ v } \vdash x \text{ p } \nearrow \rho'} \quad \rho(x) = (\text{con}, -)$$

rekursive Zerlegung von Wert und Muster und anschließender Musterabgleich