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

Skript zur Vorlesung
Informatik I
Wintersemester 2006

Kapitel 9: Pattern Matching

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

Skript © 2006 Christian Böhm

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

Inhalt

1. Zweck des Musterangleichs
2. Prinzip des Musterangleichs
3. Musterangleich und statische Typprüfung - Angleichfehler zur Laufzeit
4. Das Wildcard - Muster von SML
5. Das Verbund - Wildcard - Muster von SML
6. Die gestuften Muster von SML
7. Linearitätsbedingung für Muster
8. Der Musterangleichsalgorithmus

Überblick

1. Zweck des Musterangleichs
2. Prinzip des Musterangleichs
3. Musterangleich und statische Typprüfung - Angleichfehler zur Laufzeit
4. Das Wildcard - Muster von SML
5. Das Verbund - Wildcard - Muster von SML
6. Die gestuften Muster von SML
7. Linearitätsbedingung für Muster
8. Der Musterangleichsalgorithmus

Muster in Wertdeklarationen (1)

- Der *Musterangleich* (*Pattern Matching*) dient zur Selektion der Komponenten eines zusammengesetzten Wertes und kann in Wertdeklarationen so verwendet werden:

```
- val paar = (1.52, 4.856);
val paar = (1.52,4.856) : real * real

- val (komp1, komp2) = paar;
val komp1 = 1.52 : real
val komp2 = 4.856 : real
```

Der Ausdruck `(komp1,komp2)` stellt ein „*Muster*“ (*Pattern*) dar, das dazu dient, eine Gestalt festzulegen. In dieser Deklaration kann das Muster `(komp1,komp2)` mit dem Wert des Namens `paar`, d.h. mit dem Vektor `(1.52,4.856)` „*angeglichen*“ werden. Dieser „*Angleich*“ (*Matching*) liefert Bindungen für die Namen `komp1` und `komp2`.

Muster in Wertdeklarationen (2)

- Ist der Angleich unmöglich, so scheitert die Wertdeklaration.
- Beispiel:

```
- val (k1, k2, k3) = (1.0, 2.0);
Error: pattern and expression in val dec don´t agree
[tycon mismatch]
    pattern: 'Z * 'Y * 'X
    expression: real * real
    in declaration:
          (k1,k2,k3) =
                (case (1.0,2.0)
                      of (k1,k2,k3) => (k1,k2,k3))
```


Muster zur Fallunterscheidung in Funktionsdefinitionen (1)

- Muster und die Technik des Musterangleichs werden oft in Fallunterscheidungen verwendet.
- Beispiel einer polymorphen Funktionsdefinition (vgl. Kapitel 5.5):

```
- fun laenge nil = 0
    |  laenge(h :: L) = 1 + laenge(L);
val laenge = fn : 'a list -> int

- laenge [0,1,2,3];
val it = 4 : int
```

- Die Muster `nil` und `(h :: L)` decken alle Listenarten ab:
  - Das Muster `nil` deckt den Fall der leeren Liste ab
  - Das Muster `(h :: L)` deckt den Fall aller nichtleeren Listen ab.

Muster zur Fallunterscheidung in Funktionsdefinitionen (2)

- Die statische Typprüfung von SML erkennt, ob die verschiedenen Fälle einer Funktionsdeklaration etwaige Fälle nicht berücksichtigen.
- Beispiel einer polymorphen Listenfunktion (vgl. Kapitel 5.5):

```
- fun letztes_element(x :: nil) = x
    | letztes_element(h :: L) = letztes_element(L);
Warning: match nonexhaustive
         x :: nil => ...
         h :: L => ...
val letztes_element = fn : 'a list -> 'a

- letztes_element [1,2,3,4];
val it = 4 : int
```

Warnung in `letztes_element`

- Bei der Warnung handelt es sich um KEINEN Fehler.
- Der Fall „leere Liste“ fehlt in der Definition der Funktion `letztes_element`, weil eine leere Liste kein letztes Element haben kann.
- Die Typprüfung des SML-Systems kennt die Semantik der Funktion nicht.
- Sie erkennt nur, dass die Fallunterscheidung den Fall „leere Liste“ nicht abdeckt und macht den Programmierer darauf aufmerksam, der die Semantik der Funktion kennt (oder kennen sollte.)

head

- Muster und Musterangleich werden verwendet, wenn Funktionen, insbesondere Selektoren, über zusammengesetzten Typen, u.a. über rekursiven Typen, definiert werden.
- Die polymorphen Selektoren `head` (in SML: `hd`) und `tail` (in SML: als `tl`) für Listen sind Beispiele dafür:

```
- fun head(x :: _) = x;
Warning: match nonexhaustive
          x :: _ => ...
val head = fn : 'a list -> 'a

- head [1,2,3];
val it = 1 : int
```

tail

```
- fun tail(_ :: L) = L;
Warning: match nonexhaustive
          _ :: L => ...
val tail = fn : 'a list -> 'a list

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

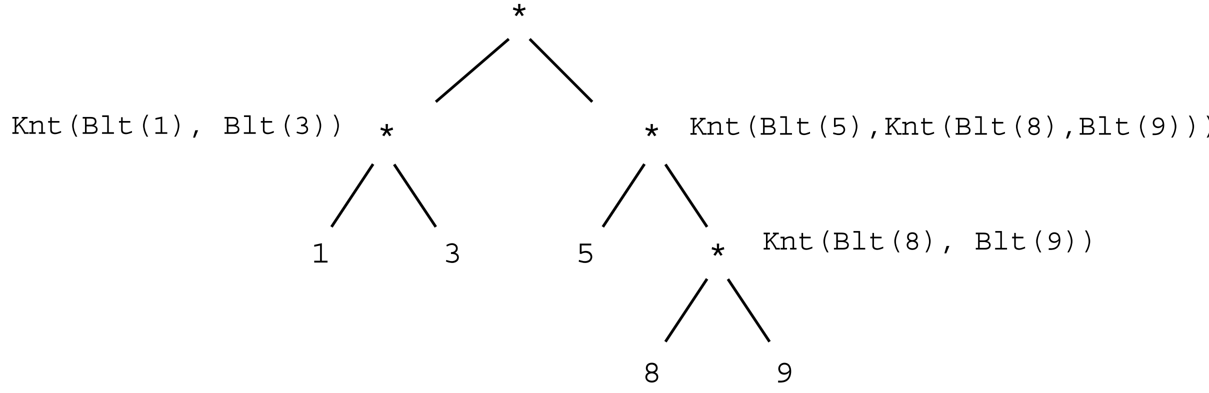
Beispiel `binbaum1`

- Auch im folgenden Beispiel werden Muster und Musterangleich verwendet, um eine Funktion über zusammengesetzten Werten zu deklarieren (es handelt sich um einen selbst definierten rekursiven Typ):

```
- datatype binbaum = Leer
                   | Blt of int
                   | Knt of binbaum * binbaum;
datatype binbaum = Blt of int
                 | Knt of binbaum * binbaum
                 | Leer;

- 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 #)) : binbaum
```


Graphische Darstellung des Binärbaums



blaetterzahl

```
- fun    blaetterzahl Leer = 0
     |   blaetterzahl (Blt(_)) = 1
     |   blaetterzahl (Knt(b1,b2)) = blaetterzahl b1 +
                                      blaetterzahl b2;
val blaetterzahl = fn : binbaum -> int

- blaetterzahl(b);
val it = 5 : int
```

- In dieser Definition der Funktion `blaetterzahl` kommt eine Fallunterscheidung vor, deren drei Fälle durch Muster entsprechend der Definition des Typs `binbaum` definiert sind.

Definition von Funktionen über Typen mit endlichen Wertemengen

- Muster und die Technik des Musterangleichs bieten sich an zur Definition von Funktionen über Typen mit endlichen Wertemengen.
- Für jeden der endlich vielen Werte kann man einen Fall angeben.
- Beispiel:
  Die Implementierung einer logischen Disjunktion `OR`.

OR

```
- fun OR(true, true) = true
    | OR(true, false) = true
    | OR(false, true) = true
    | OR(false, false) = false;
val OR = fn : bool * bool -> bool

OR(OR(false, true), true);
val it = true : bool
```

> Die Auswertungs-reihenfolge hängt ausschließlich davon ab, dass `OR` eine Funktion ist, und nicht davon, ob diese Funktion mit Hilfe von Mustern definiert ist oder auf andere Weise!

- Zur Auswertung eines Ausdrucks der Gestalt `OR(A,A´)` wertet SML immer beide Argumente von `OR` aus, während bei einem Sonderausdruck `A orelse A´` der Teilausdruck `A`, nicht immer ausgewertet wird.

Muster zur Fallunterscheidung in `case` - Ausdrücken

- Muster und die Technik des Musterangleichs können auch in herkömmlichen Ausdrücken verwendet werden.
- Beispiel:
  Man kann die Funktion `OR` neu definieren.

OR´

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

- OR´(OR´(false, true), true);
val it = true : bool
```

In SML werden auch bei dieser Definition beide Argumente ausgewertet.
Diese Frage hängt wie bereits erwähnt davon ab, ob ein Ausdruck ein *Sonderausdruck* oder eine *Funktionsanwendung* ist, aber nicht davon, mit welchen Hilfsmitteln die Funktion definiert ist.

`case`: Basis vieler SML - Implementierungen

- Die Behandlung von Mustern in `case`-Ausdrücken ist in vielen SML-Implementierungen die Basis, auf die alle anderen Verwendungen von Mustern intern zurückgeführt werden.

Beispiel

- Fehlermeldung bei einer Wertdeklaration, in der Muster und Wert nicht zusammenpassen:

```
- val (k1, k2, k3) = (1.0, 2.0);
Error: pattern and expression in val dec don´t agree
  [tycon mismatch]
      pattern: ‘Z * ‘Y * ‘X
      expression: real * real
      in declaration:
             (k1,k2,k3) =
                    (case (1.0,2.0)
                          of (k1,k2,k3) => (k1,k2,k3))
```

Überblick

1. Zweck des Musterangleichs
2. Prinzip des Musterangleichs
3. Musterangleich und statische Typprüfung - Angleichfehler zur Laufzeit
4. Das Wildcard - Muster von SML
5. Das Verbund - Wildcard - Muster von SML
6. Die gestuften Muster von SML
7. Linearitätsbedingung für Muster
8. Der Musterangleichsalgorithmus

Angleichregel

- Eine sogenannte *Angleichregel* ist ein Ausdruck der folgenden Gestalt:

  < *Muster* > => < *Ausdruck* >

- In `fun` - Deklarationen sind Angleichregeln „versteckt“, die ersichtlich werden, wenn die `fun` - Deklarationen durch ihre Bedeutung als `val` - Ausdrücke ersetzt werden.

Beispiel

- Die `fun` - Deklaration der Funktion `OR` (vgl. Abschnitt 9.1):

```
- fun OR(true, true) = true
    |  OR(true, false) = true
    |  OR(false, true) = true
    |  OR(false, false) = false;
val OR = fn : bool * bool -> bool
```

steht für die folgende `val` – Deklaration (mit Angleichregeln):

```
- val rec OR = fn (true, true) => true
                | (true, false) => true
                | (false, true) => true
                | (false, false) => false;
val OR = fn : bool * bool -> bool
```

Prüfung einer Angleichregel gegen einen Wert

- Eine Angleichregel < *Muster* > => < *Ausdruck* > wird gegen einen Wert `W` geprüft:
  - `W` wird mit dem Muster < *Muster* > angeglichen.
  - Gelingt der Angleich, so werden etwaige Namen, die im Muster vorkommen, gebunden.

Angleich (1)

- Der Angleich erfolgt dadurch:
  - dass die Strukturen von Muster und Wert gleichzeitig durchlaufen oder zerlegt werden und
  - dass dabei die Teilausdrücke komponentenweise und rekursiv angeglichen werden.
- Die Basisfälle dieses rekursiven Algorithmus liefern Namen und Konstanten, die im Muster vorkommen.
- Ein in einem Muster vorkommender Name kann, so lange die Typen übereinstimmen, mit jedem Wert angeglichen werden.
- Wird ein Name, der im Muster vorkommt, an einen Wert angeglichen, so wird der Wert an diesen Namen gebunden.
- Eine in einem Muster vorkommende Konstante kann nur mit derselben Konstante angeglichen werden.

Angleich (2)

- Konstanten und Namen stellen die einfachsten Muster dar (was die Gestalt angeht).
- Wie der Angleich genau durchgeführt wird, wird am Ende dieses Kapitels erläutert.

Prüfung eines Angleichmodells gegen einen Wert

- Angleichregeln können zu einem Angleichmodell zusammengesetzt werden:

  < *Angleichregel* > | ...| < *Angleichregel* >
- Die Angleichregeln werden sequenziell durchlaufen, wenn ein Wert `W` gegen sie geprüft wird.
- Ist der Angleich von `W` mit dem Muster einer Angleichregel des Angleichmodells erfolgreich, so wird `W` gegen die nachfolgenden Angleichregeln nicht geprüft.

Typkorrektheit eines Angleichmodells

- Ein Angleichmodell der Gestalt

  Muster$_1$ => Ausdruck$_1$ | Muster$_2$ => Ausdruck$_2$ |…| Muster$_n$ => Ausdruck$_n$

  ist nur dann korrekt typisiert, wenn:

  1. Muster$_1$, Muster$_2$,…, Muster$_n$ alle denselben Typ haben, und
  2. Ausdruck$_1$, Ausdruck$_2$,…, Ausdruck$_n$ alle denselben Typ haben.

- Die Muster können einen anderen Typ haben als die Ausdrücke.
- Diese Bedingungen können während der statischen Typprüfung überprüft werden.

Herkömmliche Angleichmodelle in SML

- Aus den bisher beschriebenen Prinzipien ergeben sich die beiden herkömmlichen Formen von Angleichmodellen in SML:

  `case` < *Ausdruck* > `of` < *Angleichmodell* >

  und

  `fn` < *Angleichmodell* >

- Erinnerung:

  Die `fun` - Deklarationen mit Musterangleich sind „syntaktischer Zucker“ des vorangehenden Angleichmodells mit `fn` - Ausdrücken.

Überblick

1. Zweck des Musterangleichs
2. Prinzip des Musterangleichs
3. Musterangleich und statische Typprüfung - Angleichfehler zur Laufzeit
4. Das Wildcard - Muster von SML
5. Das Verbund - Wildcard - Muster von SML
6. Die gestuften Muster von SML
7. Linearitätsbedingung für Muster
8. Der Musterangleichsalgorithmus

Musterangleich und statische Typprüfung (1)

- Während der statischen Typprüfung kann ein Angleichmodell bereits auf Typkorrektheit überprüft werden.
- Beispiel:

```
- fun f (0, true) = true
   | f (1, 2) = false;
Error: parameter or result constraints of clauses don´t
  agree [literal]
      this clause: int * int -> 'Z
      previous clauses: int * bool -> 'Z
      in declaration:
              f =
                (fn (0,true) => true
                      | (1,2) => false)
```

Musterangleich und statische Typprüfung (2)

- Auch unterschiedliche Typen bei Ausdrücken in einem Angleichmodell werden statisch erkannt:

```
- fun f true = 1
  | f false = 0.0;
Error: right-hand-side of clause doesn´t agree
with function result type [literal]
  expression: real
  result type: int
  in declaration:
      f =
           (fn true => 1
            | false => 0.0)
```


Angleichfehler zur Laufzeit

- Da Angleichmodelle, die nicht alle Fälle abdecken, möglich sind, kann es vorkommen, dass Angleichfehler erst zur Laufzeit erkannt werden können statt schon zur Übersetzungszeit:

```
- fun f(0) = 0
    | f(x) = if x > 0 then f(x-1) else f(~x-1);
val f = fn : int -> int
- fun g(1) = true;
Warning: match nonexhaustive
          1 => ...
val g = fn : int -> bool
- g(f(1));
uncaught exception nonexhaustive match failure
```

Überblick

1. Zweck des Musterangleichs
2. Prinzip des Musterangleichs
3. Musterangleich und statische Typprüfung - Angleichfehler zur Laufzeit
4. Das Wildcard - Muster von SML
5. Das Verbund - Wildcard - Muster von SML
6. Die gestuften Muster von SML
7. Linearitätsbedingung für Muster
8. Der Musterangleichsalgorithmus

Eigenschaften des Wildcard - Musters

- Das sogenannte *Wildcard - Muster* von SML („ _ “) hat zwei Eigenschaften:
  - Es kann an jeden Wert angeglichen werden.
  - Es wird bei einem erfolgreichen Angleich an keinen Wert gebunden.
- Typischerweise wird es als *Fangfall* verwendet.

Beispiel: Fangfall

- Betrachte folgende Funktionsdeklaration (vgl. Kapitel 2.5):

```
val Ziffer = fn  0 => true
               | 1 => true
               | 2 => true
               | 3 => true
               | 4 => true
               | 5 => true
               | 6 => true
               | 7 => true
               | 8 => true
               | 9 => true
               | _ => false;
```

Beispiel

- Wie die folgende Funktionsdeklaration (vgl. Kapitel 5.5) zeigt, darf das Wildcard - Muster auch in einem zusammengesetzten Ausdruck vorkommen, um an Teilausdrücke angeglichen zu werden, die im definierenden Teil der Funktionsdefinition nicht von Belang sind:

```
- fun head(x :: _) = x;
Warning: match nonexhaustive
          x :: _ => ...
val head = fn : 'a list -> 'a

- head([1,2,3]);
val it = 1 : int
```

Überblick

1. Zweck des Musterangleichs
2. Prinzip des Musterangleichs
3. Musterangleich und statische Typprüfung - Angleichfehler zur Laufzeit
4. Das Wildcard - Muster von SML
5. **Das Verbund - Wildcard - Muster von SML**
6. Die gestuften Muster von SML
7. Linearitätsbedingung für Muster
8. Der Musterangleichsalgorithmus

Erinnerung: Verbunde

- Verbunde sind zusammengesetzte Strukuren, deren Komponenten Namen tragen.
- Beispiel aus Kapitel 5.3:

```
- val adressbucheintrag = {Nachname = "Meier",
                           Vornamenbuchstabe = #"F",
                           Durchwahl = "2210"};
val adressbucheintrag ={Durchwahl = "2210",
                        Nachname = "Meier",
                        Vornamenbuchstabe = #"F"}
: {Durchwahl:string,Nachname:string,Vornamenbuchstabe:char}
- adressbucheintrag = {Vornamenbuchstabe = #"F",
                       Durchwahl = "2210",
                       Nachname = "Meier"};
val it = true : bool
```

- Für einen Verbund ist die Reihenfolge der Verbundkomponenten irrelevant.

Verbund - Wildcard – Muster

- Die vollständige Angabe eines Verbundes verlangt oft viel Schreibarbeit.
- Mit dem *Verbund - Wildcard - Muster* erleichtert SML dem Programmierer oder dem Leser eines Programms die Arbeit:

```
- val {Nachname = NN, ...} = adressbucheintrag;
val NN = "Meier" : string
```

- Der Name `NN` kann so an den Wert `"Meier"` (vom Typ Zeichenfolge) gebunden werden, ohne dass das Muster, in dem der Name `NN` vorkommt, die Struktur des Verbundes vollständig angibt.

Beispiele

```
- val v1 = {a = 1, b = 2};
val v1 = {a=1,b=2} : {a:int, b:int}

- val v2 = {a = 0, b = 2, c = 3};
val v2 = {a=1,b=2,c=3} : {a:int, b:int, c:int}

- val {a = N, ...} = v1;
val N = 1 : int

- val {a = N, ...} = v2;
val N = 0 : int
```

- In beiden Wertdeklarationen, die ein Verbund - Wildcard - Muster enthalten, ist eindeutig, an welchen Wert der Name `N` gebunden werden soll.

Voraussetzung

- Eine Verwendung des Verbund - Wildcard - Muster setzt aber voraus, dass der Typ des Verbundes, in dem das Verbund - Wildcard - Muster vorkommt, statisch (zur Übersetzungszeit) bestimmt werden kann.
- Bei der folgenden Funktionsdeklaration ist das nicht möglich. Deshalb wird ein Fehler gemeldet:

```
- fun f({a = 1, ...}) = true
    | f(_) = false;
Error: unresolved flex record
(can´t tell what fields there are besides #a)
```


Überblick

1. Zweck des Musterangleichs
2. Prinzip des Musterangleichs
3. Musterangleich und statische Typprüfung - Angleichfehler zur Laufzeit
4. Das Wildcard - Muster von SML
5. Das Verbund - Wildcard - Muster von SML
6. Die gestuften Muster von SML
7. Linearitätsbedingung für Muster
8. Der Musterangleichsalgorithmus

Gestufte Muster (1)

- SML bietet die sogenannten „*gestuften Muster*“ (*layered pattern*) an, die am folgenden Beispiel eingeführt werden können:

```
- val info1dozent = {Dozent = ("Boehm",#"C"),
                     Raum = "1.58"};
val info1dozent = {Dozent ("Boehm", #"C"), Raum = "1.58"}
: {Dozent:string * char, Raum:string}
- val {Dozent = D as (N,V), Raum = R} = info1dozent;
val D = ("Boehm", #"C") : string * char
val N = "Boehm" : string
val V = #"C" : char
val R = "1.58" : string
```


Gestufte Muster (2)

- Mit dem Konstrukt

  `D as (N,V)`

  erfolgen neben
  - der Bindung des Wertes `("Boehm",#"C"),` an den Namen `D`
  - die Bindungen des Wertes `"Boehm"` an den Namen `N` und
  - die Bindung des Wertes `#"C"` an den Namen `V`.

Überblick

1. Zweck des Musterangleichs
2. Prinzip des Musterangleichs
3. Musterangleich und statische Typprüfung - Angleichfehler zur Laufzeit
4. Das Wildcard - Muster von SML
5. Das Verbund - Wildcard - Muster von SML
6. Die gestuften Muster von SML
7. Linearitätsbedingung für Muster
8. Der Musterangleichsalgorithmus

Linearitätsbedingung für Muster

- Die Verwendung von Mustern und der Technik des Musterangleichs verlangen, dass die Muster in dem Sinne *linear* sind, dass eine ungebundene Variable, die durch den Angleich des Musters an einen Wert gebunden werden soll, nur einmal - daher die Bezeichnung *linear* - im Muster vorkommt.

Beispiel

- Die folgende polymorphe Funktion

```
- fun gleich(x, y) = (x = y);
val gleich = fn : ''a * ''a -> bool
- gleich([1,2], [1,2]);
val it = true : bool
- gleich(#"a", #"a");
val it = true : bool
```

kann NICHT wie folgt deklariert werden:

```
- val gleich = fn (x, x) => true
                | _      => false;
Error: duplicate variable in pattern(s): x
```


Grund

- Der Grund für diese Einschränkung ist, den Pattern-Matching-Algorithmus einfach und folglich effizient zu halten.
- Auch die Typprüfung wäre ohne diese Einschränkung betroffen, weil Mehrfachvorkommen von Variablen in Mustern zur Folge haben müssten, dass die Werte an den entsprechenden Positionen zu Gleichheitstypen gehören müssen.

Überblick

1. Zweck des Musterangleichs
2. Prinzip des Musterangleichs
3. Musterangleich und statische Typprüfung - Angleichfehler zur Laufzeit
4. Das Wildcard - Muster von SML
5. Das Verbund - Wildcard - Muster von SML
6. Die gestuften Muster von SML
7. Linearitätsbedingung für Muster
8. **Der Musterangleichsalgorithmus**

Informelle Spezifikation des Musterangleichsalgorithmus

- Erinnerung an Kapitel 9.2:
  - Der Angleich (zwischen einem Muster und Wert) erfolgt dadurch:
    - dass die Strukturen von Muster und Wert gleichzeitig durchlaufen oder zerlegt werden und
    - dass dabei die Teilausdrücke komponentenweise und rekursiv angeglichen werden.
  - Die Basisfälle dieses rekursiven Algorithmus liefern Namen und Konstanten, die im Muster vorkommen.
  - Ein in einem Muster vorkommender Name kann, so lange die Typen übereinstimmen, mit jedem Wert angeglichen werden.
  - Wird ein Name, der im Muster vorkommt, an einen Wert angeglichen, so wird der Wert an diesen Namen gebunden.
  - Eine in einem Muster vorkommende Konstante kann nur mit derselben Konstante angeglichen werden.

Ausdruck vs. Wert im Musterangleich

- Der Musterangleich erfolgt zwischen einem Muster `M` und einem Wert `W`.
- In einem Programm sind aber jeweils ein Muster `M` und ein Ausdruck `A` gegeben.
- Beispiel:

  `val M = A` oder

  `case A of M =>`
- Bei der Auswertung wird zunächst `A` in der aktuellen Umgebung ausgewertet zu einem Wert `W`.
- Danach wird der Musterangleichsalgorithmus auf das Muster `M` und diesen Wert `W` angewandt.

Umgebung (Wiederholung aus Kapitel 2)

- Erinnerung an Kapitel 2.7:
  - Das SML-System verwaltet mit jeder Sitzung und jeder eingelesen Datei eine geordnete Liste der Gestalt Name=Wert (dargestellt als Paare (Name,Wert)). Diese Liste heißt *Umgebung*.
  - Jede neue Deklaration eines Wertes `W` für einen Namen `N` führt zu einem neuen Eintrag `N=W` am Anfang der Umgebung.
  - Um den Wert eines Namens zu ermitteln, wird die Umgebung von Anfang an durchlaufen. So gilt immer als Wert eines Namens `N` derjenige Wert, der als letztes angegeben wurde.
- Der Musterangleichsalgorithmus muss sein Ergebnis in einer Form liefern, die für die Erweiterung der aktuellen Umgebung um neue Einträge geeignet ist.

Formale Spezifikation des Musterangleichsalgorithmus (1)

- Der Musterangleichsalgorithmus wird auf ein Muster `M` und einen Wert `W` angewandt.
- Der Musterangleichsalgorithmus soll, wenn möglich, `M` und `W` angleichen und dabei Werte für die Namen, die im Muster `M` vorkommen, ermitteln.
- Die Bindung dieser Werte an diese Namen erfolgt nicht *während* des Musterangleichs, sondern erst *danach*.
- D.h. wenn der Musterangleichsalgorithmus die aktuelle Umgebung nicht verändert, dann liefert er nur die Bindungen, um die die Umgebung anschließend erweitert werden kann.

Der Einfachheit halber werden im folgenden Algorithmus die gestuften Muster und das Verbund-Wildcard-Muster nicht berücksichtigt (die entsprechende Erweiterung des Algorithmus stellt aber keine große Schwierigkeit dar).

Formale Spezifikation des Musterangleichsalgorithmus (2)

- Der Musterangleichsalgorithmus soll auch feststellen, dass der Angleich von Muster `M` und Wert `W` unmöglich ist.
- In dem Fall sagt man, dass der Angleich von `M` und `W` gescheitert ist.
- Der Musterangleichsalgorithmus meldet also:
  - Einen Erfolg und liefert eine (endliche) Menge von Bindungen für die (endlich vielen) Namen, die im Muster `M` vorkommen (dabei bleibt die Umgebung unverändert).

    Jede dieser Bindungen wird dargestellt als ein Paar `(Name, Wert)`.
  - Ein Scheitern.

Musterangleichsalgorithmus (1)

- Zur Angleichung eines Musters `M` und eines Werts `W` gehe so vor:

1. Falls `M` eine Konstante `k` ist, dann:
   - (a) Falls `W` ebenfalls die Konstante `k` ist, dann: liefere die leere Menge von Bindungen und terminiere erfolgreich.
   - (b) Andernfalls terminiere erfolglos.
2. Falls `M` ein Name ist, dann: liefere die einelementige Menge { `(M,W)` } von Bindungen und terminiere erfolgreich.
3. Falls `M` das Wildcard - Muster ist, dann: liefere die leere Menge von Bindungen und terminiere erfolgreich.

(Fortsetzung nächste Folie)

Musterangleichsalgorithmus (2)

4. Falls `M` zusammengesetzt ist mit (Wert-) Konstruktor `K` und Teilmustern $M_1, \ldots, M_n$, dann:
   - (a) Falls `W` ebenfalls zusammengesetzt ist mit demselben (Wert-) Konstruktor `K` und Teilwerten $W_1, \ldots, W_n$, dann: wende für jedes $i \in \{1,\ldots,n\}$ den Musterangleichsalgorithmus auf das Muster M_i und den Wert W_i an.
     - i. Falls eine dieser Anwendungen des Musterangleichs-algorithmus scheitert, dann terminiere erfolglos.
     - ii. Andernfalls bilde die Vereinigung aller Bindungsmengen, die diese Anwendungen des Musterangleichsalgorithmus liefern; liefere die so erhaltene Menge von Bindungen und terminiere erfolgreich.
   - (b) Andernfalls terminiere erfolglos.

Beispiel einer Anwendung des Musterangleichsalgorithmus (1)

- `M` sei das Muster `e1 :: (e2 :: _ ),`
  `W` sei der Wert `1 :: (2 :: (3 :: (4 :: (5 :: nil)))):`
- Fall 4: `M` ist zusammengesetzt und hat die Gestalt `M1 K M2`:
  - `K` ist der Konstruktor `::`,
  - `M1` ist das Muster `e1`,
  - `M2` ist das Muster `e2 ::_`
- Fall 4(a): `W` ist zusammengesetzt und hat die Gestalt `W1 K W2`:
  - `W1` ist der Wert `1`,
  - `W2` ist der Wert `2 :: (3 :: (4 :: (5 :: nil)))`

Beispiel einer Anwendung des Musterangleichsalgorithmus (2)

- Anwendung des Musterangleichsalgorithmus auf `M1` und `W1`:
  - Nebenrechnung, in der gilt:
    - `M` ist das Muster `e1`,
    - `W` ist der Wert `1`

    Fall 2: `M` ist ein Name:
    - liefere die Menge { `(e1,1)` } und terminiere erfolgreich
  - Ende der Nebenrechnung mit Erfolg, Ergebnis { `(e1,1)` }
- Anwendung des Musterangleichsalgorithmus auf `M2` und `W2`:
  - Nebenrechnung, in der gilt:
    - `M` ist das Muster `e2 :: _`,
    - `W` ist der Wert `2 :: (3 :: (4 :: (5 :: nil)))`

Beispiel einer Anwendung des Musterangleichsalgorithmus (3)

Fall 4: `M` ist zusammengesetzt und hat die Gestalt `M1 K M2`:
- `K` ist der Konstruktor `::`,
- `M1` ist das Muster `e2`,
- `M2` ist das Muster _

Fall 4(a): `W` ist zusammengesetzt und hat die Gestalt `W1 K W2`:
- `W1` ist der Wert `2`,
- `W2` ist der Wert `3 :: (4 :: (5 :: nil))`
- Anwendung des Algorithmus auf `M1` und `W1`:
  - Nebenrechnung, in der gilt:
    - `M` ist das Muster `e2`,
    - `W` ist der Wert `2`

    Fall 2: `M` ist ein Name:
    - liefere die Menge { `(e2,2)` } und terminiere erfolgreich
  - Ende der Nebenrechnung mit Erfolg, Ergebnis { `(e2,2)` }

Beispiel einer Anwendung des Musterangleichsalgorithmus (4)

- Anwendung des Algorithmus auf `M2` und `W2`:
  - Nebenrechnung, in der gilt:
    - `M` ist das Muster _,
    - `W` ist der Wert `3 :: (4 :: (5 :: nil))`

    Fall 3: `M` ist das Wildcard - Muster :
    - liefere die leere Menge und terminiere erfolgreich.
  - Ende der Nebenrechnung mit Erfolg, Ergebnis { }

Fall 4(a)ii: Beide Anwendungen waren erfolgreich:
- Bilde die Vereinigung von { `(e2,2)` } und { };
- liefere { `(e2,2)` } und terminiere erfolgreich

- Ende der Nebenrechnung mit Erfolg, Ergebnis { `(e2,2)` }

Beispiel einer Anwendung des Musterangleichsalgorithmus (5)

- Fall 4(a)ii: Beide Anwendungen waren erfolgreich:
  - Bilde die Vereinigung von { `(e1,1)` } `und` { `(e2,2)` };
  - liefere { `(e1,1),(e2,2)` } und terminiere erfolgreich.
- Ende der gesamten Berechnung, Erfolg, Ergebnis `(e1,1),(e2,2).`

Korrektheit und Terminierung des Musterangleichsalgorithmus

- Die Korrektheit des Musterangleichsalgorithmus bedeutet:
  - Wenn der Musterangleichsalgorithmus eine Menge von Bindungen liefert, dann ergibt eine Ersetzung der Namen im Muster durch die Werte, die die Bindungen diesen Namen zuordnen, genau den Wert, mit dem das Muster angeglichen wurde.
  - Wenn der Musterangleichsalgorithmus ein Scheitern meldet, dann gibt es keine Bindungen mit dieser Eigenschaft.
- Diese Aussage lässt sich durch strukturelle Induktion beweisen, wobei für jeden möglichen (Wert-) Konstruktor ein Induktionsfall nötig ist.

Terminierung

- Die Terminierung lässt sich ebenfalls durch strukturelle Induktion beweisen.
- Entscheidend dabei sind die folgenden Beobachtungen:
  - Die Fälle 1, 2 und 3 des Musterangleichsalgorithmus, die nicht zusammengesetzte Muster behandeln, terminieren offensichtlich.
  - Die Terminierung des Falles 4 des Musterangleichsalgorithmus, der zusammengesetzte Muster behandelt, wird induktiv bewiesen. Dabei sind die Induktionsannahmen, dass die `n` Anwendungen des Musterangleichsalgorithmus auf M_i und W_i für jedes $i \in \{1,\ldots,n\}$ terminieren.

Folgen der Linearätsbedingung für den Musterangleichsalgorithmus (1)

- Die Linearitätsbedingung für Muster macht es möglich, dass während einer Anwendung des Musterangleichsalgorithmus die erzeugten Bindungen unbesehen in die Ergebnismenge übernommen werden können.
- Der Ablauf des Musterangleichsalgorithmus hängt an keiner Stelle davon ab, welche Bindungen erzeugt wurden.
- Insbesondere sind die Bindungsmengen, die im Schritt 4(a)ii vereinigt werden, garantiert disjunkt, so dass die Vereinigung durch ein triviales Aneinanderhängen implementiert werden kann.

Folgen der Linearätsbedingung für den Musterangleichsalgorithmus (2)

- Ohne die Linearitätsbedingung könnten Namen mehrfach vorkommen, so dass die `n` Bindungsmengen, die im Schritt 4(a)ii vereinigt werden, mehrere Bindungen für denselben Namen enthalten könnten, die außerdem teilweise gleich und teilweise verschieden sein könnten.
- Bei gleichen Bindungen für einen Namen müssten die Mehrfachvorkommen erkannt und entfernt werden.
- Bei verschiedenen Bindungen für einen Namen müssten diese erkannt und ein Scheitern gemeldet werden.
- Das ist zwar alles implementierbar, aber nur mit höherer Komplexität des Algorithmus!

Folgen der Linearätsbedingung für den Musterangleichsalgorithmus (3)

- Der Ablauf des Musterangleichsalgorithmus hängt überhaupt nicht mehr von anderen Daten ab als vom Muster, das bereits zur Übersetzungszeit bekannt ist - ohne die Linearitätsbedingung würden die rekursiven Aufrufe auch von den Bindungen abhängen, die erst zur Laufzeit bekannt sind.
- Wenn die Linearitätsbedingung eingehalten wird, kann man die rekursiven Aufrufe bereits zur Übersetzungszeit „entfalten“ in eine einzige Schachtelung von `if-then-else`.
- Die Struktur dieser Schachtelung hängt nur vom Muster ab.
- Ob diese Optimierung von SML-Implementierungen tatsächlich durchgeführt wird, ist damit nicht festgelegt, entscheidend ist nur, dass durch die Entscheidung für die Linearitätsbedingung beim Design der Programmiersprache diese Optimierung ermöglicht wurde.

Folgen der Linearätsbedingung für den Musterangleichsalgorithmus (4)

- Neuere funktionale Sprachen mit Pattern Matching verlangen die Linearitätsbedingung noch aus einem anderen Grund:

  Sie haben wesentlich differenziertere Gleichheitstypen als SML, mit denen man zum Beispiel jeweils eigene Gleichheitsprädikate verbinden kann.
- Dann wären Mehrfachvorkommen einer Variablen in einem Muster nur sinnvoll, wenn jeweils das zugehörige Gleichheitsprädikat mit angegeben würde, wodurch die Syntax von Mustern praktisch unlesbar würde.
- Die Linearitätsbedingung für Muster hat also auch den Vorteil, die Verwendung von Mustern für Programmierer einfacher und übersichtlicher zu machen.