Damit die angegebene Transformation sinnvoll definiert ist, müssen wir vermeiden, dass funktionale Muster zyklisch definiert werden, d.h. bei der Auswertung eines funktionalen Musters darf nicht die Funktion, die damit definiert wird, selbst ausgewertet werden. Z.B. ist die Regel

```
(xs ++ ys) ++ zs = xs ++ (ys ++ zs)
```

unzulässig, weil die Bedeutung des funktionalen Musters `(xs ++ ys)` von der Bedeutung der Funktion "`++`" abhängt. Formal kann man dies mit Ebenenabbildungen (level mappings) festlegen:

Definition 4.14 (Ebenenabbildung) *Eine **Ebenenabbildung** (**level mapping**) m für ein Programm P ist eine Abbildung von Funktionsnamen, die in P definiert sind, zu natürlichen Zahlen, sodass für alle Regeln $f\ t_1 \ldots t_n \mid c = e$ gilt: falls g eine Funktion ist, die in c oder e vorkommt, dann gilt $m(g) \leq m(f)$.*

Wenn z.B. `last` durch die Regel

```
last :: [a]  → a
last xs | _++[x] =:= xs
        = x
   where x free
```

definiert ist, dann ist $m($`++`$) = 0$ and $m($`last`$) = 1$ eine mögliche Ebenenabbildung. Größere Werte einer Ebenenabbildung bedeuten somit, dass diese Funktionen auf der Definition von Funktionen mit kleineren Werte aufbauen. Mittels Ebenenabbildungen können wir nun zulässige Programme definieren.

Definition 4.15 (Stratifizierte Programme) *Ein logisch-funktionales Programm P mit funktionalen Mustern ist **stratifiziert** (*stratified*), falls eine Ebenenabbildung m für P existiert, sodass für alle Regeln $f\ t_1 \ldots t_n \mid c = e$ gilt: falls eine definierte Funktion g in einem Muster t_i vorkommt ($i \in \{1, \ldots, n\}$), dann gilt $m(g) < m(f)$.*

Die Restriktion auf stratifizierte Programme garantiert, dass, falls eine Operation f mit einem funktionalen Muster p definiert ist, die Auswertung von p zu einem Konstruktorterm weder direkt noch indirekt von f abhängt.

Bevor wir auf einige Beispiele zur Benutzung funktionaler Muster eingehen, wollen wir kurz deren Implementierung diskutieren, die erstaunlich einfach ist. Hierzu wird zusätzlich zur Standardunifikation "`=:=`" eine erweiterte Unifikation "`=:<=`" zur Implementierung funktionaler Muster eingeführt. Ein funktionales Muster wird hierbei ersetzt durch einen Aufruf dieser erweiterten Unifikation, wobei das linke Argument das funktionale Muster ist. Zum Beispiel wird die Regel

```
last (xs++[x]) = x
```

zur Übersetzungszeit durch die neue Regel

```
last ys | xs++[x] =:<= ys  = x
   where xs,x free
```

ersetzt.

Die erweiterte Unifikation "`=:<=`" wird analog zur Standardunifikation "`=:=`" abgearbeitet (vgl. Kapitel 4.6), allerdings wird zunächst nur das linke Argument zur Kopfnormalform ausgewertet. Die operationale Bedeutung von "`=:<=`" kann durch folgende Metadefinition zur Auswertung von "e_1 `=:<=` e_2" angegeben werden:

1. Werte e_1 zur Kopfnormalform h_1 aus.
2. Falls h_1 eine Variable ist: binde diese an e_2, d.h. das Ergebnis ist die Substitution $\{h_1 \mapsto e_2\}$ und der Wert `True`.
3. $h_1 = C\ t_1 \ldots t_n$ (wobei C ein Konstruktor ist): Werte e_2 zur Kopfnormalform h_2 aus:
   - Falls h_2 eine Variable ist: binde diese an $C\ y_1 \ldots y_n$ (wobei $y_1, \ldots, y_n$ neue Variablen sind) und werte dann den Ausdruck

     t_1`=:<=`y_1 `&` $\ldots$ `&` t_n`=:<=`y_n

     aus.
   - Falls $h_2 = C\ s_1 \ldots s_n$: werte den Ausdruck

     t_1`=:<=`s_1 `&` $\ldots$ `&` t_n`=:<=`s_n

     aus.
   - Sonst: Fehlschlag

Der entscheidende Unterschied zur Standardunifikation "`=:=`" ist also die zunächst zurückgestellte Auswertung des rechten Arguments (der aktuelle Parameter des Funktionsaufrufes).[12]. Falls das linke Argument, also das funktionale Muster, eine Variable ist, wird dieses an das rechte (unausgewertete!) Argument gebunden, ansonsten wird wie üblich weitergemacht, d.h. das rechte Argument wird zur Kopfnormalform ausgewertet und eine Fallunterscheidung (Konstante, Konstruktor, Variable) gemacht. Durch das Binden einer Variablen, die in einem funktionalen Muster vorkommt, an einen *unausgewerteten* aktuellen Parameter, wird der entscheidende Vorteil funktionaler Muster erreicht.

Betrachten wir als Beispiel die Auswertung von "`last [failed,2]`". Mittels der transformierten Regel und der erweiterten Unifikation "`=:<=`" erhalten wir folgende erfolgreiche

[12] Auf Grund dieser nicht-strikten Auswertung wird diese Unifikation manchmal auch als nicht-strikte Unifikation bezeichnet.

Berechnungsfolge:

```
last [failed,2]  ⇝                (xs++[x] =:<= [failed,2]) &> x
                 ⇝{xs↦x1:xs1}     (x1:(xs1++[x]) =:<= [failed,2]) &> x
                 ⇝                (x1 =:<= failed & xs1++[x] =:<= [2]) &> x
                 ⇝{x1↦failed}     (xs1++[x] =:<= [2]) &> x
                 ⇝{xs1↦[]}        ([x] =:<= [2]) &> x
                 ⇝                (x =:<= 2 & [] =:<= []) &> x
                 ⇝{x↦2}           ([] =:<= []) &> 2
                 ⇝                True &> 2
                 ⇝                2
```

Durch diese Implementierung werden konzeptionell die unendlichen vielen Muster, die durch ein funktionales Muster beschrieben werden, zur Laufzeit bedarfsgesteuert berechnet. Falls also das aktuelle Argument nur eine endliche Größe hat (d.h. keine unendliche Datenstruktur ist), dann werden typischerweise auch nur endliche viele Muster aus dem funktionalen Muster zur Unifikation berechnet.

Bevor wir weitere Beispiele betrachten, wollen wir noch auf ein potenzielles Problem funktionaler Muster eingehen. Ein funktionales Muster kann eventuell zu einem nichtlinearen Term ausgewertet werden. Betrachten wir hierzu

```
idpair x = (x,x)

f (idpair x) = 0
```

Nach der Semantik funktionaler Muster ist die Definition von `f` äquivalent zu

```
f (x,x) = 0
```

Solche Muster sind im Gegensatz zu Haskell in Curry erlaubt (und man könnte deren Vorkommen statisch auch nicht verbieten). Wie schon in in Kapitel 4.1 auf Seite 110 erwähnt wurde, werden mehrfache Vorkommen von Variablen in Mustern als Gleichheitsbedingungen interpretiert, sodass die Definition von `f` letztendlich äquivalent zu

```
f (x,y) | x =:= y = 0
```

ist. Dies bedeutet, dass z.B. `f (x,failed)` keinen Wert hat. Somit müssen also für Variablen, die im Ergebnis funktionaler Muster mehrfach vorkommen, noch Gleichheitsconstraints erzeugt werden, was in der obigen Beschreibung aus Vereinfachungsgründen weggelassen wurde. Die Implementierung hiervon ist nicht-trivial, weil diese Eigenschaft nur zur Laufzeit festgestellt werden kann. Z.B. kommt in den Definitionen

```
zero x = 0

pair x y = (x,y)

f (pair (zero x) x) = 0
```

die Variable `x` im funktionalen Muster von `f` mehrfach vor, aber nach der Definition funktionaler Muster ist die Definition von `f` äquivalent zu

```
f (0,x) = 0
```

Somit muss in diesem Fall kein Gleichheitsconstraint für `x` generiert werden. Diese Beispiele zeigen, dass für eine korrekte Implementierung funktionaler Muster die Vorkommen von Variablen in den zur Laufzeit erzeugten Mustern kontrolliert werden muss.

Weil durch funktionale Muster Gleichheitsconstraints erzeugt werden *könnten*, gelten für diese die gleichen Entschränkungen wie für die Unifikation, d.h. funktionale Muster haben immer einen `Data`-Kontext (vgl. Kapitel 4.6.3) und die nicht-strikte Unifikation hat den Typ

```
(=:<=) :: Data a => a  → a  → a
```

Im Folgenden wollen wir einige Beispiele für funktionale Muster betrachten. Wie wir schon bei der Definition von `last` gesehen haben, sind funktionale Muster nützlich, um „tiefe“ Muster in Listen zu beschreiben. Damit können wir z.B. auch sehr einfach Permutationen definieren:

```
perm :: Data a => [a]  → [a]
perm []           = []
perm (xs ++ x:ys) = x : perm (xs ++ ys)
```

Ebenso können wir nicht aufsteigend sortierte Listen charakterisieren, indem wir prüfen, ob zwei benachbarte Elemente in absteigender Ordnung existieren:

```
someDescending :: [Int]  → Bool
someDescending (_++x:y:_) | x > y = True
```

Damit können wir die Sortierung einer Liste charakterisieren: eine Permutation, die `someDescending` *nicht* erfüllt. Die letztere Eigenschaft können wir mittels Mengenfunktionen definieren (ähnlich wie beim n-Damen Problem in Kapitel 4.7.2), sodass wir folgende Definition erhalten:

```
sort :: [Int]  → [Int]
sort xs | isEmpty (set1 someDescending p) = p
  where p = perm xs
```

Als weiteres einfaches Beispiel wollen wir eine Funktion

```
lengthUpToRepeat :: Eq a => [a]  → Int
```

definieren, die die *Länge einer Liste bis zum ersten wiederholten Element* berechnet (dies war Teil eines ACM-Programmierwettbewerbes). So soll beispielsweise

```
lengthUpToRepeat [1,2,3,42,56,2,3,1]
```

das Ergebnis `6` liefern. Wir müssen also die Liste so aufteilen, dass in dem ersten Teil keine doppelten Elemente sind und dann ein Element folgt, das im ersten Teil vorkommt. Mittels funktionaler Muster ist dies einfach definierbar:

```
lengthUpToRepeat (p ++ [r] ++ q)
    | nub p == p && r `elem` p
    = length p + 1
```

Man beachte, dass wir diese Operation auch auf unendliche Listen anwenden können, z.B. wird der Ausdruck

```
lengthUpToRepeat ([1,2,3,42,56] ++ [0..])
```

zu `7` ausgewertet. Ohne funktionale Muster, d.h. mit der Standardunifikation, wäre dies nicht möglich, da die Unifikation

```
(p ++ [r] ++ q) =:= ([1,2,3,42,56] ++ [0..])
```

wegen der strikten Gleichheit nicht terminiert.

Funktionale Muster sind nützlich, wenn komplexe Transformationen oder Suchen in Termen realisiert werden sollen. Betrachten wir hierzu *symbolische arithmetische Ausdrücke*, wie z.B. $1 * (x + 0)$. Die Aufgabe ist es, solche Ausdrücke zu vereinfachen, in diesem Fall also zu x. Hierzu definieren wir einen Datentyp für arithmetische Ausdrücke:

```
data Exp = Lit Int
         | Var String
         | Add Exp Exp
         | Mul Exp Exp
```

Somit kann also $1 * (x + 0)$ als Term

```
(Mul (Lit 1) (Add (Var "x") (Lit 0)))
```

dargestellt werden. Da wir Ausdrücke vereinfachen wollen, müssen wir Teilausdrücke in einem Ausdruck ersetzen. Für diese Ersetzung definieren wir eine Operation `replace`, sodass (`replace` e p t) der Ersetzung $e[t]_p$ in der Notation der Termersetzungssysteme entspricht:

```
replace :: Exp  → [Int]  → Exp  → Exp
replace _         []    x = x
replace (Add l r) (1:p) x = Add (replace l p x) r
replace (Add l r) (2:p) x = Add l (replace r p x)
replace (Mul l r) (1:p) x = Mul (replace l p x) r
replace (Mul l r) (2:p) x = Mul l (replace r p x)
```

Wir benötigen noch eine Darstellung der einzelnen Simplifikationsregeln. Hierzu verwenden wir eine Definitionsart, die für funktionale Muster oft nützlich ist. Wir definieren eine

nichtdeterministische Operation, die zu einem Ausdruck mögliche „gleiche“ Ausdrücke liefert:

```
evalTo :: Exp  → Exp
evalTo e = Add (Lit 0) e
         ? Add e (Lit 0)
         ? Mul (Lit 1) e
         ? Mul e (Lit 1)
```

Diese Definition kann natürlich noch um weitere Möglichkeiten ergänzt werden. Hiermit ist die Definition der Vereinfachung wie folgt möglich:

```
simplify :: Exp  → Exp
simplify (replace c p (evalTo x)) = replace c p x
```

Wir vereinfachen einen Ausdruck also, indem wir einen Teilausdruck durch einen semantisch gleichen Teilausdruck ersetzen. Somit wird z.B.

```
simplify (Mul (Lit 1) (Var "y"))
```

zu `(Var "y")` ausgewertet. Den obigen Beispielausdruck können wir durch zweifache Anwendung von `simplify` vereinfachen, d.h.

```
simplify (simplify (Mul (Lit 1) (Add (Var "x") (Lit 0))))
```

wird zu `(Var "x")` ausgerechnet. Beliebige Simplifikationen könnte man mittels eingekapselter Suche berechnen.

Dieses Beispiel zeigt zwei Anwendungen funktionaler Muster. Einerseits wurden Kollektionen von Mustern zusammengefasst (mittels `evalTo`), andererseits können wir das Pattern Matching eines Teilausdrucks an einer beliebig tief geschachtelten Position (der Wert für die Position `p` wird dabei geraten) einfach realisieren. Als weiteres Beispiel für die letzte Technik können wir z.B. Variablennamen in einem Term einfach auffinden:

```
varInExp :: Exp  → String
varInExp (replace _ _ (Var v)) = v
```

Mittels eingekapselter Suche können wir auch alle Variablennamen finden. Z.B. liefert der Ausdruck

$\texttt{varInExp}_{\mathcal{S}}\ e$

die Menge aller Variablennamen des Ausdrucks e.