Der Satz von Hullot stellt also die Forderungen der Konfluenz und Terminierung an ein Termersetzungssystem:

1. Konfluenz: sinnvoll (vgl. funktionale Programmierung)
2. Terminierung: Dies stellt eine Einschränkung dar:
   - Wie kann man die Terminierung überprüfen?
   - Dies erlaubt keine unendlichen Datenstrukturen.

Narrowing ist aber auch vollständig bei nichtterminierenden Systemen, allerdings nur für **normalisierte Substitutionen** (d.h. in 2. muss σ' normalisiert sein, d.h. $\sigma'(x)$ ist in Normalform $\forall x \in Dom(x)$).

Das folgende Beispiel zeigt die Unvollständigkeit von Narrowing bei nicht-normalisierten Substitutionen:

$$\begin{array}{rcl} f(x,x) & \to & 0 \\ g & \to & c(g) \end{array}$$

Betrachten wir die Gleichung:

$$f(y, c(y)) \doteq 0$$

Hier ist kein Narrowing-Schritt möglich, aber $\{y \mapsto g\}$ ist eine Lösung, denn

$$f(g, c(g)) \to f(c(g), c(g)) \to 0$$

Wenn man allerdings unendliche Datenstrukturen zulässt, ergibt sich das Problem, wie man den Gültigkeitsbegriff der Gleichheit so sinnvoll definieren kann, dass man dies auch effektiv überprüfen kann. Bisher ist die Gleichheit $\doteq$ reflexiv, d.h. $t \doteq t$ ist immer gültig für alle Terme t. Aus diesem Grund wird „$\doteq$“ auch als **reflexive Gleichheit** bezeichnet.

Betrachten wir nun die Definitionen

$$\begin{array}{rcl} f & \to & 0 : f \\ g & \to & 0 : g \end{array}$$

D.h. sowohl f als auch g sind unendliche Listen mit 0 als Element. Auf Grund der reflexiven Gleichheit ist

$$f \doteq f$$

gültig. Da f und g identisch definiert sind, müsste dann auch

$$f \doteq g$$

gültig sein. Aber wie soll man diese Gleichheit, d.h. die Gleichheit unendlicher Objekte, im Allg. überprüfen? Man kann dann leicht Fälle konstruieren, bei denen eine solche Überprüfung unentscheidbar ist.

Betrachten wir weiterhin die Definitionen

$$\begin{array}{lcl} h(x) & \to & h(x) \\ k(x) & \to & k(x) \end{array}$$

Ist nun

$$h(0) \doteq k(0)$$

gültig? Beide Berechnungen sind „gleich“, denn sie terminieren nicht!

Als Konsequenz dieser Beispiele kann man ableiten, dass bei nichtterminierenden Termersetzungssystemen die reflexive Gleichheit nicht sinnvoll ist. Aus diesem Grund verwendet man in logisch-funktionalen (wie auch in funktionalen) Programmiersprachen die **strikte Gleichheit**, die wir mit "`=:=`" bezeichnen.
Intuitiv bezeichnet `=:=` die Gleichheit auf endlichen Strukturen:

Definition 4.4 *Die **strikte Gleichheit** t_1 `=:=` t_2 ist gültig, falls t_1 und t_2 zu einem **Grundkonstruktorterm** u reduzierbar sind, d.h. es gilt $t_1 \to^* u$ und $t_2 \to^* u$ wobei u keine Variablen und keine definierten Funktionen enthält.*

Obige Beispiele: Die Gleichheiten f `=:=` f, f `=:=` g und $h(0)$ `=:=` $k(0)$ sind nicht gültig.

Die strikte Gleichheit `=:=` entspricht `==`, wobei man hier nur an positiven Ergebnissen (`True`) interessiert ist. Durch diese Einschränkung auf positive Ergebnisse kann man bestimmte Berechnungen auch effizienter durchführen, wie wir noch sehen werden.

Ein interessanter Aspekt hierbei ist, dass die strikte Gleichheit `=:=` ist (im Gegensatz zu $\doteq$) durch ein funktionales Programm definierbar (ebenso wie "`==`", wobei hier die `False`-Regeln weggelassen werden):

$$\begin{array}{rcll} c \texttt{ =:= } c & \to & \texttt{True} & \text{(für alle 0-stelligen Konstruktoren } c\text{)} \\ d(x_1,\ldots,x_n) \texttt{ =:= } d(y_1,\ldots,y_n) & \to & x_1 \texttt{ =:= } y_1 \texttt{ \&\& } \ldots \texttt{ \&\& } x_n \texttt{ =:= } y_n & \\ & & & \text{(für alle n-stelligen Konstruktoren } d\text{)} \\ \texttt{True \&\& } x & \to & x & \end{array}$$

Dann gilt:

Satz 4.2 *s `=:=` t ist gültig $\Leftrightarrow$ s `=:=` $t \to^*$ `True` mittels obiger Zusatzregeln.*

Beachte: Die reflexive Gleichheit kann man nicht entsprechend definieren, da

$$x \doteq x \to \texttt{True}$$

eine unzulässige Regel in Haskell ist (weil die linke Seite nicht linear ist).

4.3 Spezialfall: Logikprogrammierung

Wie wir in der Einleitung schon gesehen haben, gibt es zwei wichtige Klassen deklarativer Sprachen:

- Funktionale Sprachen
- Logische Sprachen

Beide können als Spezialfälle *logisch-funktionaler Sprachen* (d.h. funktionale Programmierung mit freien Variablen und Nichtdeterminismus) gesehen werden:

Funktionale Sprachen:	keine freien Variablen erlaubt
Logische Sprachen:	keine (geschachtelten) Funktionen erlaubt, nur Prädikate

Da wir funktionale Programme schon genauer kennengelernt haben, wollen wir im Folgenden den Spezialfall der Logikprogramme genauer betrachten. Ein **Logikprogramm** hat folgende Eigenschaften:

- Alle Funktionen haben den Ergebnistyp `Bool` und liefern nur den Wert `True`, d.h. es sind nur positive Aussagen möglich. Solche Funktionen nennt man auch **Prädikate** oder **Constraints**.
- Alle Regeln haben die Form

  l `= True`

  beziehungsweise

  l `|` c `= True`

  Es werden also nur positive Aussagen definiert und c ist eine Konjunktion von Prädikaten.

  Da alle Regeln "`= True`" auf der rechten Seiten haben, könnte man dies syntaktisch vereinfachen, indem man die rechte Seite weglässt. Dies wird in der Programmiersprache Prolog auch gemacht, sodass man in Prolog

  $p(t_1, \ldots, t_n)$.

  statt

  $p\ t_1 \ldots t_n$ `= True`

  schreibt. In ähnlicher Form werden Regeln in Prolog aufgeschrieben:

  l'_0 `:-` $l'_1, \ldots, l'_n$.

  Dies entspricht in Curry der bedingten Regel

l_0 `|` l_1 `&&...&&` l_n `= True`

Hierbei ist l'_i von der Form $p_i(t_{i_1}, \ldots, t_{i_{n_i}})$, falls l_i die Form $p_i\ t_{i_1} \ldots t_{i_{n_i}}$ hat. Diese Form der l'_i werden auch **Literale** genannt.

- Insbesondere gibt es in der Logikprogrammierung folgende Erweiterungen gegenüber rein funktionalen Sprachen wie Haskell:
  - *Extravariablen* sind in Bedingungen erlaubt und müssen nicht deklariert werden.
  - *Nichtlineare linke Seiten* (z.B. "`eq(X,X).`") sind in Prolog zulässig.

Obwohl wir **Extravariablen** in Regeln schon erläutert haben, wollen wir darauf noch einmal explizit eingehen.

Extravariablen sind in Prolog wegen der flachen Struktur der Bedingungen unbedingt notwendig.

Beispiel: Statt

```
[]     ++ ys = ys
(x:xs) ++ ys = x:(xs++ys)

rev []     = []
rev (x:xs) = rev xs ++ [x]
```

muss man in Prolog Folgendes schreiben (in Prolog werden Variablen groß geschrieben und man schreibt "`[X|Xs]`" anstatt "`X:Xs`"):

```
append([],     Ys, Ys).   % Zusatzargument für das Ergebnis
append([X|Xs], Ys, [X|Zs]) :- append (Xs, Ys, Zs).

rev([],[]).
rev([X|Xs], Ys) :- rev(Xs, Rs), append(Rs, [X], Ys).
% Hier ist Rs eine Extravariable zum "Durchreichen" des Ergebnisses
```

Damit haben wir auch eine Methode zur Übersetzung von Funktionen in Prädikate gesehen:

- Führe ein Zusatzargument für das Ergebnis einer Funktion ein, d.h. übersetze eine Funktion

  f `::` $t_1 \to \ldots \to t_n$

  in ein Prädikat mit dem Typ

  f `::` $t_1 \to \ldots \to t_n \to$ `Bool`

- „Abflachen“ („flattening“) von geschachtelten Aufrufen, indem diese in Konjunktionen übersetzt werden (beachte, dass hierdurch die Möglichkeit der lazy-Auswertung verloren geht!).

Praktisch verwendbar sind Extravariablen z.B. bei transitiven Abschlüssen. Wir betrachten hierzu als Beispiel die Definition einer Vorfahrrelation in unserem Verwandtschaftsbeispiel (vgl. Kap. 4.1):

```
vorfahr v p | v == mutter p                  = True
vorfahr v p | v == vater  p                  = True
vorfahr v p | v == mutter p1 && vorfahr p1 p = True   where p1 free
vorfahr v p | v == vater  p1 && vorfahr p1 p = True   where p1 free
```

In den letzten beiden Regeln ist eine Extravariable `p1` notwendig.

Bedingte Regeln

- Bedingte Regeln sind für Logikprogramme wichtig: falls man nur unbedingte Regeln der Form

  $p\ t_1 \ldots t_n$ `= True`

  zulassen würde, wäre das Programm nicht mehr als eine Menge von Fakten einer relationalen Datenbank.

- Rechnen mit bedingten Regeln der Form

  l `|` c `=` r

  Falls l „passt“ (bzgl. pattern matching oder Unifikation), dann berechne c:
  Falls c reduzierbar zu `True`, dann ist das Ergebnis der Regelanwendung r, ansonsten ist diese Regel nicht anwendbar (d.h. probiere eine andere Regel).

- Diese Interpretation bedingter Regeln ist leicht formalisierbar durch eine Transformation: übersetze

  l `|` c `=` r

  in eine „unbedingte“ Regel der Form

  l `= cond` c r

  wobei die Operation `cond` definiert ist durch:

  ```
  cond True x = x
  ```

  Beispiel: Die Funktion `f` sei definiert durch die bedingten Regeln

```
f x | x==0 = 0
f x | x==1 = 1
```

Diese werden übersetzt in

```
f x = cond (x==0) 0
f x = cond (x==1) 1
```

Zu beachten ist, dass diese Art der Übersetzung nur funktioniert, falls alle Regeln gleichzeitig (nichtdeterministisch) angewendet werden! Diese Übersetzung ist also nicht für Haskell geeignet, da die transformierten Regeln evtl. nicht konfluent sind.

Mit dieser Übersetzung ist dann z.B. die folgende Reduktion möglich:

`f 0` $\rightarrow$ `cond (0==0) 0` $\rightarrow$ `cond True 0` $\rightarrow$ `0`

Logikprogramme haben also die folgenden Eigenschaften:

- Sie enthalten i.d.R. freie Variablen, d.h. sie müssen durch Narrowing (und nicht durch Reduktion) abgearbeitet werden.
- Sie bestehen aus Regeln der Form

  l_0 `|` l_1 `&&...&&` l_n `= True`

  Für diese Art von Regeln können wir die "`cond`"-Transformation vereinfachen zu

  l_0 `=` l_1 `&&...&&` l_n

  wobei wir die folgende Regel für `&&` verwenden können:

  ```
  True  && x    = x
  ```

Wenn unsere Programme nur diese einfache Form von Regeln haben, dann vereinfacht sich das allgemeine Narrowing-Verfahren bei Logikprogrammen zu dem bei der Logikprogrammierung eingesetzten Resolutionsprinzip:

Definition 4.5 (Resolution) *Gegeben sei eine Konjunktion (**Anfrage**, **Ziel**, **goal**) der Form*

l_1 `&&...&&` l_n

Falls $l_0 = r_0$ (hierbei ist r_0 entweder `True` oder eine Konjunktion) eine Variante einer Regel ist und σ ein mgu für l_i ($i > 0$) und l_0, dann ist

l_1 `&&...&&` l_n $\quad\leadsto_\sigma\quad$ $\sigma(l_1$ `&&...&&` l_{i-1} `&&` r_0 `&&` l_{i+1} `&&...&&` $l_n)$

*ein **Resolutionsschritt**.*

Somit:

1. Unifiziere beliebiges Literal in der Anfrage mit der linken Seite einer Regel.
2. Falls erfolgreich: Ersetze dieses Literal durch die rechte Seite der Regel und wende den Unifikator an.

Beispiel:

```
p a = True
p b = True
q b = True
```

Anfrage: `p x && q x` $\overset{\texttt{x}\mapsto\texttt{a}}{\rightsquigarrow}$ `True && q a` $\overset{\{\}}{\rightsquigarrow}$ `q a` : Fehlschlag, Sackgasse
$\overset{\texttt{x}\mapsto\texttt{b}}{\rightsquigarrow}$ `True && q b` $\overset{\{\}}{\rightsquigarrow}$ `q b` $\overset{\{\}}{\rightsquigarrow}$ `True`

Wie an diesem Beispiel zu sehen ist, ist es zur Lösungsfindung notwendig, **alle** Regeln auszuprobieren. In den meisten existierenden Logiksprachen wird dieses Ausprobieren durch ein Backtracking-Verfahren wie folgt realisiert:

Backtracking:

- Probiere zunächst die erste passende Regel aus und rechne weiter.
- Falls diese Berechnung in eine Sackgasse führt: Probiere (bei der letzten Alternative) die nächste passende Regel aus und rechne damit weiter.

Das potenzielle Problem bei dem Backtracking-Verfahren ist, dass man evtl. keine Lösung bei unendlichen Berechnungen findet.

Beispiel:

```
p a = p a
p b = True
```

Anfrage: `p x` $\overset{\texttt{x}\mapsto\texttt{a}}{\rightsquigarrow}$ `p a` $\rightarrow$ `p a` $\rightarrow$ `p a` $\rightarrow$ ...

Somit haben wir hier eine Endlosschleife, sodass die Lösung $\{\texttt{x} \mapsto \texttt{b}\}$ nicht gefunden wird.

Dieses Beispiel erscheint hier trivial, aber tatsächlich ist dies ein häufiges Problem bei der Programmierung mit rekursiven Datenstrukturen:

```
last (x:xs) e  =  last xs e
last [x]    e  | x == e = True
```

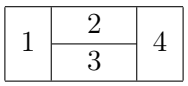
Betrachten wir folgende Anfrage:

```
last xs 0 ⇝{xs↦(x1:xs1)}  last xs1 0
          ⇝{xs1↦(x2:xs2)} last xs2 0
          ⇝ ...
```

Wir erkennen, dass man bei der Suche nach potenziell unendlichen Strukturen also vorsichtig sein muss!

Bei endlichen Strukturen kann die Suche eine sinnvolle Programmiertechnik sein. Betrachten wir hierzu das Beispiel des Färbens einer Landkarte:

Gegeben sei die folgende einfache Landkarte mit vier Ländern:



Die einzelnen Länder sollen mit Rot, Gelb, Grün, Blau eingefärbt werden, sodass aneinandergrenzende Länder verschiedene Farben haben. Wir können dieses Problem wie folgt lösen:

1. Aufzählung der Farben:

```
data  Farbe = Rot | Gelb | Gruen | Blau
 deriving (Eq,Show)

farbe Rot   = True
farbe Gelb  = True
farbe Gruen = True
farbe Blau  = True
```

2. Mögliche Färbung der Karte:

```
faerbung l1 l2 l3 l4 = farbe l1 && farbe l2 && farbe l3 && farbe l4
```

3. Korrekte Färbung: Aneinandergrenzende Länder haben verschieden Farben:

```
korrekt l1 l2 l3 l4 =
  l1 /= l2 && l1 /= l3 && l2 /= l3 && l2 /= l4 && l3 /= l4
```

Eine Lösung können wir nun berechnen, indem wir die Konjunktion aus der Aufzählung möglicher Lösungen (`faerbung l1 l2 l3 l4`) und dem Prüfen der Korrektheit einer Lösung (`korrekt l1 l2 l3 l4`) bilden:

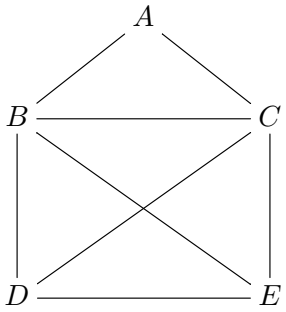
```
solve $ faerbung l1 l2 l3 l4 && korrekt l1 l2 l3 l4  where l1,l2,l3,l4 free
↝ {l1=Rot, l2=Gelb, l3=Gruen, l4=Rot} True
↝ {l1=Rot, l2=Gelb, l3=Gruen, l4=Blau} True
⋮
```

Diese Lösung basiert auf der Programmiertechnik „**Aufzählung des Suchraumes**“ („**generate-and-test**“). Das allgemeine Schema hierbei ist:

generate s `&&` *test* s `where` s `free`

Dieses Schema ist sinnvoll, falls *generate* s endlich viele (und möglichst wenige) Möglichkeiten für s berechnet.

Wir betrachten noch ein weiteres Beispiel, bei der die Suche stärker mit der Berechnung verwoben ist. Die Aufgabe ist, das bekannte „Haus vom Nikolaus“ in einem Zug zu zeichnen:



Es sollen also alle dargestellten Kanten in einem Zug gezeichnet werden. Hierzu definieren wir zunächst die Knoten:

```
data Node = A | B | C | D | E
 deriving (Eq,Show)
```

Eine Kante besteht einfach aus zwei Knoten:

```
data Edge = Edge Node Node
 deriving Eq
```

Damit können wir alle vorhandenen Kanten als folgende Liste darstellen:

```
allEdges = [Edge A B, Edge A C, Edge B C, Edge B D, Edge B E,
            Edge C D, Edge C E, Edge D E]
```

Um nun alle diese Kanten zu zeichnen und dabei eine Knotenfolge zu besuchen, definieren wir ein Prädikat `draw`, das eine Liste von Kanten und Knoten als Argumente hat:

```
draw :: [Edge]  →  [Node]  →  Bool
```

Das Prädikat “`draw edges nodes`” soll erfüllt sein, wenn man alle Kanten aus `edges` so anordnen kann, dass diese in einem Zug die Knoten aus `nodes` „besuchen“. Wenn also `nodes` mindestens zwei Knoten enthält, müssen wir eine Kante zwischen diesen Knoten finden und mit den restlichen Kanten die restlichen Knoten besuchen. Enthält `nodes` nur noch einen Knoten, dann müssen auch alle Kanten aufgebraucht sein. Somit erhalten wir die folgende Definition:

```
draw edges (n1:n2:nodes) | chooseEdge n1 n2 edges == rest
                         = draw rest (n2:nodes) where rest free
draw [] [_] = True -- keine Kanten, nur ein Knoten
```

Hierbei sucht `chooseEdge` eine Kante zwischen den gegebenen Knoten heraus und gibt die restlichen Kanten zurück. Zu beachten ist nur, dass Kanten in beiden Richtungen benutzt werden können, sodass wir zwei Regeln haben, die erste Kante zu verwenden, und eine weitere Regel zur Benutzung einer anderen Kante haben:

```
chooseEdge :: Node → Node → [Edge] → [Edge]
chooseEdge n1 n2 (Edge n1 n2 : edges) = edges
chooseEdge n1 n2 (Edge n2 n1 : edges) = edges
chooseEdge n1 n2 (edge : edges)       = edge : chooseEdge n1 n2 edges
```

Hierbei haben wir wieder die Eigenschaft von Curry ausgenutzt, dass man auch Muster definieren kann, bei denen Variablen mehrfach vorkommen. Dies ist allerdings nur syntaktischer Zucker für eine Gleichheitsbedingung zwischen diesen Vorkommen, d.h. die erste Regel für `chooseEdge` ist eine Abkürzung für die folgende Regel mit einem linearen Muster:

```
chooseEdge n1 n2 (Edge m n : edges) | m==n1 && n==n2 = edges
```

Nun können wir eine Lösung für das Kantenzeichnen finden, indem wir die Operation `draw` mit allen Kanten aber einer unbekannten Knotenliste aufrufen:

```
draw allEdges nodes where nodes free
↝ {nodes=[D,B,A,C,B,E,C,D,E]} True
↝ {nodes=[D,B,A,C,B,E,D,C,E]} True
⋮
```