Notizen zur Vorlesung

Deklarative Programmiersprachen

Sommersemester 2014

Prof. Dr. Michael Hanus

Arbeitsgruppe Programmiersprachen und Übersetzerkonstruktion
Christian-Albrechts-Universität zu Kiel

Version vom 31. Juli 2014

Vorwort

Durch die Komplexität heutiger Software-Systeme ist die Verwendung von Programmiersprachen mit einem hohen Abstraktionsniveau notwendig. Deklarative Sprachen bieten hierzu wichtige Lösungsansätze. Auf Grund ihrer deklarativen Struktur sind die Programme leichter wartbar und verifizierbar. In dieser Vorlesung werden Konzepte moderner deklarativer Programmiersprachen vorgestellt.

Ausgehend von dem aus dem Grundstudium bekannten Konzept der funktionalen Programmierung, das kurz wiederholt und eingehender erläutert wird, werden funktionale Sprachen um logische Anteile erweitert, um die Konzepte der funktionalen, logischen und integrierten logisch-funktionalen Sprachen in einem einheitlichen Rahmen darzustellen. Außerdem werden Grundlagen der funktionalen und logischen Programmierung vorgestellt.

Dieses Skript gibt einen Überblick über die in der Vorlesung behandelten Inhalte. Leichte Abweichungen sind möglich, sodass es nicht als Ersatz für die Vorlesungsteilnahme dienen kann, die zum Verständnis der Konzepte und Techniken wichtig ist. Ich danke insbesondere Björn Peemöller für viele Korrekturen in diesem Skript.

Kiel, Juli 2014 Michael Hanus

P.S.: Wer in diesem Skript keine Fehler findet, hat sehr unaufmerksam gelesen. Ich bin für alle Hinweise auf Fehler dankbar, die mir persönlich, schriftlich oder per E-Mail mitgeteilt werden.

Inhaltsverzeichnis

0 Einleitung

Die Geschichte und Entwicklung von Programmiersprachen kann aufgefasst werden als Bestreben, immer stärker von der konkreten Hardware der Rechner zu abstrahieren. In der folgenden Tabelle sind einige Meilensteine dieser Entwicklung aufgeführt:

Sprachen/Konzepte:	Abstraktion von:
Maschinencode	
Assembler (Befehlsnamen, Marken)	Befehlscode, Adresswerte
Fortran: arithmetische Ausdrücke	Register, Auswertungsreihenfolge
Algol: Kontrollstrukturen, Rekursion	Befehlszähler, "goto"
Simula/Smalltalk/Java: Klassen, Objekte	Implementierung ("abstrakte Datentypen") Speicherverwaltung
Deklarative Sprachen: Spezifikation von Eigenschaften	Ausführungsreihenfolge ↪ Optimierung ↪ Parallelisierung ↪ keine Seiteneffekte

Eine der wichtigsten charakteristischen Eigenschaften deklarativer Sprachen ist die **referenzielle Transparenz** (engl. **referential transparency**):

- Ausdrücke dienen *nur* zur Wertberechnung
- Der Wert eines Ausdrucks hängt nur von der Umgebung ab und nicht vom Zeitpunkt der Auswertung
- Ein Ausdruck kann durch einen anderen Ausdruck gleichen Wertes ersetzt werden (**Substitutionsprinzip**)

Beispiel: Üblich in der Mathematik:

Wert von $x^2 - 2x + 1$ hängt nur vom Wert von x ab
$\Rightarrow$ - Variablen sind Platzhalter für Werte
- eine Variable bezeichnet immer gleichen Wert

Beispiel: Die Programmiersprache C ist nicht referenziell transparent:

- Hier sind Variablen Namen von Speicherzellen

- Deren Werte (Inhalte) können verändert werden:

  ```
  x = 3;
  y = (++x) * (x--) + x;
  ```

  Hier bezeichnen das erste und das letzte Vorkommen der Variablen `x` *verschiedene* Werte!
  Außerdem ist der Wert von `y` nach der Ausführung abhängig von der Auswertungsreihenfolge!

Substitutionsprinzip:

- natürliches Konzept aus der Mathematik
  Beispiel: Ersetze π immer durch 3,14159...
- Fundamental für Beweisführung („ersetze Gleiches durch Gleiches“, vereinfache Ausdrücke)
- vereinfacht:
  - Optimierung
  - Transformation
  - Verifikation

  von Programmen

Beispiel: *Tranformation und Optimierung*

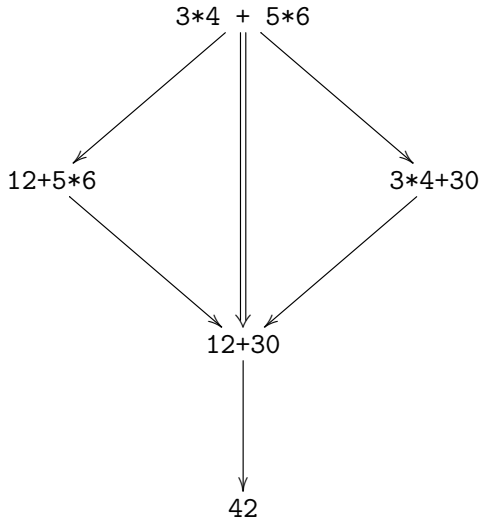
```
function f(n:nat) : nat =
begin
   write("Hallo");
   return (n * n)
end
```

`...z := f(3)*f(3) ...` $\Rightarrow$ `z=81` und `HalloHallo`

Mögliche Optimierung(!?): `...x:=f(3); z:=x*x ...`

$\Rightarrow$ **Seiteneffekte erschweren Optimierung**

Beispiel: Flexible Auswertungsreihenfolge:
Betrachten wir verschiedene Auswertungsmöglichkeiten des Ausdrucks `3*4 + 5*6`:



Hierbei bezeichnet der Doppelpfeil in der Mitte die parallele Auswertung des Ausdrucks. Durch die flexible Auswertungsreihenfolge, die durch referenzielle Transparenz ermöglicht wird, kann man somit **Parallelrechner** leichter ausnutzen.

Betrachten wir einen weiteren wichtigen Aspekt deklarativer Sprachen: Die Erstellung **lesbarer und zuverlässiger Programme**. Hierzu beginnen wir mit zwei Zitaten:

- Programme müssen so geschrieben werden, damit Menschen sie lesen und modifizieren, und nur nebenbei, damit Maschinen sie ausführen [Abelson/Sussman/Sussman 96]
- Programmierung ist eine der schwierigsten mathematischen Tätigkeiten [Dijkstra]

Lesbarkeit durch Abstraktion von Ausführungsdetails:

Beispiel: Fakultätsfunktion:
Eine Formulierung in einer imperativen Programmiersprache könnte wie folgt aussehen:

```
function fac(n:nat):nat =
begin
   z:=1; p:=1;
   while z<n+1 do
   begin p:=p*z: z:=z+1 end;
   return(p)
end
```

Potenzielle Fehlerquellen:

- Initialisierung (`z:=1` oder `z:=0`?)

- Abbruchbedingung (`z<n+1` oder `z<n` oder `z<=n`?)
- Reihenfolge im Schleifenrumpf (zuerst oder zuletzt `z:=z+1`?)

Deklarative Formulierung: Definiere Eigenschaften von `fac`:

```
fac(0)   = 1
fac(n+1) = (n+1) * fac(n)
```

⇒ kein Zähler (`z`), kein Akkumulator (`p`), Rekursion statt Schleife, weniger Fehlerquellen

Beispiel: **Quicksort**

Imperativ (nach Wirth: Algorithmen und Datenstrukturen):

```
procedure qsort (l, r: index);
var i,j: index; x,w: item
begin
   i := l; j := r;
   x := a[(l+r) div 2]
   repeat
      while a[i]<x do i := i+1;
      while a[j]>x do j := j-1;
      if i <= j then
      begin w := a[i]; a[i] := a[j]; a[j] := w;
            i := i+1; j := j-1;
      end
   until i > j;
   if l < j then qsort(l,j);
   if j > r then qsort(i,r);
end
```

Potenzielle Fehlerquellen: Abbruchbedingungen, Reihenfolge

Deklarativ (funktional): Sortieren auf Listen:
(`[]` bezeichnet die leere Liste, `(x:l)` bezeichnet eine Liste mit erstem Element `x` und Restliste `l`)

```
qsort [] = []
qsort (x : l) = qsort (filter (<x) l)
                ++ [x]
                ++ qsort (filter (>=x) l)
```

⇒ klare Codierung der Quicksort-Idee

Datenstrukturen und Speicherverwaltung

Viele Programme benötigen komplexe Datenstrukturen
Imperative Sprachen: Zeiger → fehleranfällig ("core dumped")

Beispiel: Datenstruktur **Liste**:
- entweder leer (NIL)
- oder zusammengesetzt aus 1. Element (Kopf) und Restliste

Aufgabe: Gegeben zwei Listen $[x_1, \ldots, x_n]$ und $[y_1, \ldots, y_m]$
Erzeuge Verkettung der Listen: $[x_1, \ldots, x_n, y_1, \ldots, y_m]$

Imperativ, z.B. Pascal:

```
TYPE list = ^listrec;
     listrec = RECORD elem: etype;
                      next: list
               END;

PROCEDURE append(VAR x: list; y:list);
BEGIN
   IF x = NIL
   THEN x := y
   ELSE append(x^.next, y)
END;
```

Anmerkungen:

- Implementierung der Listendefinition durch Zeiger
- Verwaltung von Listen durch explizite Zeigermanipulation
- Verkettung der Listen durch Seiteneffekt

Gefahren bei expliziter Speicherverwaltung:

- Zugriff auf Komponenten von NIL
- Speicherfreigabe von unreferenzierten Objekten:
  - keine Freigabe $\leadsto$ Speicherüberlauf
  - explizite Freigabe $\leadsto$ schwierig (Seiteneffekte!)

Deklarative Sprachen abstrahieren von Details der Speicherverwaltung (automatische Speicherbereinigung)
Obiges Beispiel in funktionaler Sprache:

```
data List = [] | etype : List

append([]  , ys) = ys
append(x:xs, ys) = x : append(xs,ys)
```

Vorteile:

- Korrektheit einsichtig durch einfache Fallunterscheidung:
  z.z.: `append`$([x_1,\ldots,x_n],[y_1,\ldots,y_m]) = [x_1,\ldots,x_n,y_1,\ldots,y_m]$
  Beweis durch Induktion über Länge der 1. Liste:
  Ind. anfang: n=0: $\underbrace{\texttt{append}([],\underbrace{[y_1,\ldots,y_m]}_{ys}) = \underbrace{[y_1,\ldots,y_m]}_{ys}}_{\text{Anwenden der ersten Gleichung}}$
  Ind.schritt: $n > 0$:
  Induktionsvoraussetzung: `append`$([x_2,\ldots,x_n],[y_1,\ldots,y_m]) = [x_2,\ldots,x_n,y_1,\ldots,y_m]$
  append $([x_1,\ldots,x_n],[y_1,\ldots,y_m])$
  = append $(x_1 : \underbrace{[x_2,\ldots,x_n]}_{xs}, \underbrace{[y_1,\ldots,y_m]}_{ys})$
  = x_1: append $([x_2,\ldots,x_n],[y_1,\ldots,y_m])$ (Anwendung der 2. Gleichung)
  = $x_1 : [x_2,\ldots,x_n,y_1,\ldots,y_m]$ (Ind. vor.)
  = $[x_1,\ldots,x_n,y_1,\ldots,y_m]$ (nach Def. von Listen)

- keine Seiteneffekte ($\rightsquigarrow$ einfacher Korrektheitsbeweis)
- keine Zeigermanipulation
- keine explizite Speicherverwaltung
- Universelle Verwendung von `append` durch generische Typen ($\rightarrow$ *Polymorphismus*):

  ```
  data List a = [] | a : List a
  ```

  (`a` ist hierbei eine Typvariable)

  $\Rightarrow$ `append` auf Listen mit beliebigem Elementtyp anwendbar (im Gegensatz zu Pascal)

Klassifikation von Programmiersprachen:

- *Imperative Sprachen:*
  - Variablen = Speicherzellen (veränderbar!)
  - Programm = Folge von Anweisungen (insbesondere: Zuweisung)
  - sequenzielle Abarbeitung, Seiteneffekte
- *Deklarative Sprachen*

- Variablen = (unbekannte) Werte
- Programm = Menge von Definitionen (+auszuwertender Ausdruck)
- potenziell parallele Abarbeitung, seiteneffekfrei
- automatische Speicherverwaltung
- weitere Klassifikation nach der zu Grunde liegenden mathematischen Theorie:

  Funktionale Sprachen

  * Funktionen, λ-Kalkül [Church 40]
  * Reduktion von Ausdrücken
  * Funktionen höherer Ordnung
  * polymorphes Typkonzept
  * "Pattern Matching"
  * "Lazy Evaluation"

  Logiksprachen

  * Relationen, Prädikatenlogik
  * Resolution, Lösen von Ausdrücken [Robinson 65]
  * logische Variablen, partielle Datenstrukturen
  * Unifikation
  * Nichtdeterminismus

Funktional-logische Sprachen: Kombination der beiden Klassen

Konkrete deklarative Sprachen:

- funktional:
  - LISP, Scheme: eher imperativ-funktional
  - (Standard) ML: funktional mit imperativen Konstrukten
  - Haskell: Standard im Bereich nicht-strikter funktionaler Sprachen
    $\rightarrow$ praktische Übungen
- logisch:
  - Prolog: ISO-Standard mit imperativen Konstrukten
  - CLP: Prolog mit Constraints, d.h. fest eingebauten Datentypen und Constraint Solver
- funktional-logisch:
  - Mercury (University of Melbourne): Prolog-Syntax, eingeschränkt
  - Oz (DFKI Saarbrücken): Logik-Syntax, nebenläufig
  - Curry: Haskell-Syntax, nebenläufig

Anwendungen deklarativer Sprachen

- prinzipiell überall, da berechnungsuniversell
- Aspekte der Softwareverbesserung:
  - Codereduktion ($\rightarrow$ Produktivität):
    - Assembler/Fortran $\approx 10/1$
    - keine wesentliche Reduktion bei anderen imperativen Sprachen
    - funktionale Sprachen/imperative Sprachen $\approx 1/(5-10)$
  - Wartbarkeit: lesbarere Programme
  - Wiederverwendbarkeit:
    - polymorphe Typen (vgl. `append`)
    - Funktionen höherer Ordnung $\rightsquigarrow$ Programmschemata

Anwendungsbereiche

- Telekommunikation:
  Erlang (nebenläufige funktionale Sprache):
  - ursprünglich: Programmierung von TK-Software für Vermittlungsstellen
  - Codereduktion gegenüber imperativen Sprachen: 10-20%
- wissensbasierte Systeme: Logiksprachen
- Planungsprobleme (Operations Research, Optimierung, Scheduling):
  CLP-Sprachen
- Transformationen: funktionale Sprachen, z.B. XSLT für XML-Transformationen

Literatur

Gute Einführungen in die funktionale Programmierung findet man in [Hudak 99, Thompson 96]. Ein Standardwerk zur logischen Programmierung mit Prolog ist [Sterling/Shapiro 94]. Überblicksartikel und Einführungen in die logisch-funktionale Programmierung findet man in [Antoy/Hanus 10, Hanus 94, Hanus 07].

1 Funktionale Programmierung

In diesem Kapitel werden wir zunächst wichtige Aspekte rein funktionaler Programmiersprachen betrachten.

1.1 Funktionsdefinitionen

Grundidee: Programm	=	Menge von Funktionsdefinitionen (+ auszuwertender Ausdruck)
Funktionsdefinition:		Funktionsname
	+	formale Parameter
	+	Rumpf (Ausdruck)
Allgemein:		$f\ x_1 \ldots x_n\ =\ e$

Ausdrücke:

- elementar: Zahlen `3`, `3.14159`
- elementare Funktion: `3+4`, `1+5*7`
- Funktionsanwendungen: $f \quad a_1 \quad \ldots \quad a_n$
  Funktion *aktuelle Parameter*
- bedingte Ausdrücke: `if` b `then` e_1 `else` e_2

Beispiel: Quadratfunktion:

```
square x = x*x
```

"`=`" steht für Gleichheit von linker und rechter Seite
$\Rightarrow$ `square 9 = 9*9 = 81`

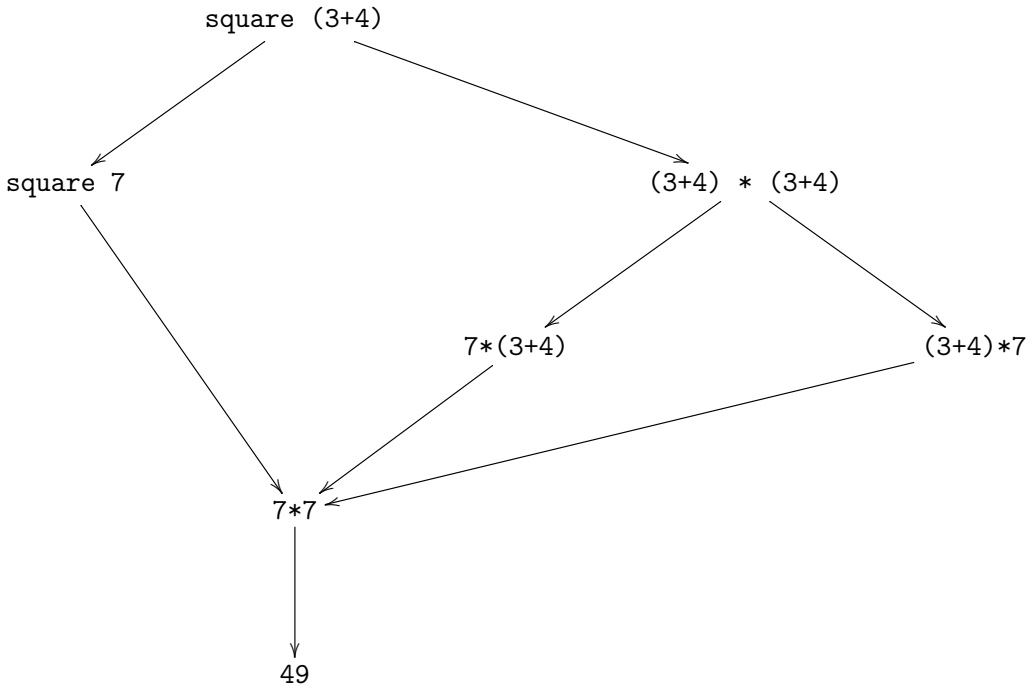
Rechnen bedeutet in deklarativen Programmiersprachen, dass Gleiches durch Gleiches ersetzt wird, bis ein Wert herauskommt. Wie funktioniert dies aber genau, wenn man mit selbst definierten Funktionen rechnet?

Auswerten von Ausdrücken:
Falls Ausdruck ein Aufruf einer elementaren Funktion (z.B. "`+`") ist und alle Argumente ausgewertet sind: Ersetze den Ausdruck durch den berechneten Wert
Sonst:

1. Suche im Ausdruck eine Funktionsanwendung (auch: **Redex**: **red**ucible **ex**pression)
2. Substitutiere in entsprechender Funktionsdefinition formale Parameter durch aktuelle (auch im Rumpf!)
3. Ersetze Teilausdruck durch Rumpf

Wegen 2: *Variablen* in Funktionsdefinitionen sind *unbekannte Werte*, aber keine Speicherzellen!
(Unterschied: imperativ $\leftrightarrow$ deklarativ)

Beispiel: Auswertungsmöglichkeiten von `square (3+4)`
(bei Anwendung elementarer Funktionen: werte vorher Argumente aus)



Jede konkrete Programmiersprache folgt einer bestimmten **Auswertungsstrategie** (vgl. Punkt 1 in obiger Beschreibung). Daher können wir folgende grobe Klassifizierung funktionaler Programmiersprachen vornehmen:

Strikte Sprachen: Wähle immer linken inneren Teilausdruck, der Redex ist (Bsp.: linker Pfad)
(*leftmost innermost, call-by-value, application order reduction, eager reduction*)

- \+ einfache effiziente Implementierung
- – berechnet evtl. keinen Wert, obwohl ein Ergebnis existiert (bei anderer Auswertung)

Nicht-strikte Sprachen Wähle immer linken äußeren Teilausdruck, der Redex ist (Bsp.: mittlerer Pfad)
(*leftmost outermost, call-by-name, normal order reduction*)
Besonderheit: Argumente sind evtl. unausgewertete Ausdrücke (nicht bei elementaren Funktionen)

+ berechnet Wert, falls einer existiert
+ vermeidet überflüssige Berechnungen
– evtl. Mehrfachauswertung (z.B. `(3+4)`)
  ⇒ Verbesserung durch *verzögerte/faule Auswertung* (*call-by-need*, *lazy evaluation*)

LISP, SML: strikte Sprachen
Haskell: nicht-strikte Sprache

Fallunterscheidung: Für viele Funktionen notwendig, z.B.

$$abs(x) = \begin{cases} x & \text{falls } x \geq 0 \\ -x & \text{sonst} \end{cases}$$

Implementierung mit if-then-else (Beachte: `abs` vordefiniert!):

```
absif x = if x >= 0 then x else -x
```

Aufruf: `absif (-5)` $\rightsquigarrow$ `5`

Implementierung mit *bedingten Gleichungen* (guarded rules)

```
absg x | x >= 0     = x
       | otherwise  = -x
```

Bedeutung der Bedingungen:
Erste Gleichung mit erfüllbarer Bedingung wird angewendet (`otherwise` $\approx$ `True`)
Diese Schreibweise entspricht daher sehr stark der mathematischen Notation.

Wir erläutern die unterschiedlichen Möglichkeiten zur Definition einer Funktion mit Fallunterscheidungen an Hand der **Fakultätsfunktion**: Generell ist diese wie folgt definiert:

$$fac(n) \quad \rightsquigarrow \quad n * (n-1) * \cdots * 2 * 1$$

Rekursive Definition:

$$fac(n) \;=\; \begin{cases} 1 & \text{falls } n = 0 \\ n * fac(n-1) & \text{falls } n > 0 \end{cases}$$

Implementierung:

1. Mit `if-then-else`:

```
fac n = if n==0 then 1 else n * fac (n-1)
```

Nicht ganz korrekt, weil rekursiver Aufruf auch bei `n<0` möglich wäre.

2. Mit bedingten Gleichungen:

```
fac n | n==0  = 1
      | n>0   = n * fac (n-1)
```

Korrekt, aber etwas umständlich.

3. **Muster** in formalen Parametern (*pattern matching*):
Bisher waren die formalen Parameter einer Funktionsdefinition nur Variablen. Nun lassen wir auch „Muster“ zu: Intuitiv ist eine Gleichung nur anwendbar, wenn der aktuelle Parameter zum Muster „passt“.

```
fac 0     = 1
fac (n+1) = (n+1) * fac n
```

Hierbei ist `(n+1)` ein Muster für alle positiven Zahlen. Allgemein steht das Muster `(n+`k`)` für ganze Zahlen $\geq k$, wobei k eine positive ganze Zahl sein muss.

Da dieses Muster später selten gebraucht wird und eher nur für die Einführung in die musterorientierte Programmierung relevant ist, wird es standardmäßig durch den Glasgow Haskell Compiler (GHC) nicht mehr unterstützt. Um es zu benutzen, muss man den GHC mit der Option “`-XNPlusKPatterns`” aufrufen zu Beginn des Quellprogramm das Pragma

```
{-# LANGUAGE NPlusKPatterns #-}
```

angeben.

Beispielberechnung:

```
fac 3                   -- 3 passt auf Muster n+1 für n=2 ⇝
 = (2+1) * fac(2)
 = 3 * fac(2)           -- 2 passt auf Muster n+1 für n=1 ⇝
 = 3 * (1+1) * fac (1)
 = 3 * 2 * fac (1)
 = 3 * 2 * (1+0) * fac (0)
 = 3 * 2 * 1 * 1   ⇝ 6
```

Musterorientierter Stil:

- kürzere Definitionen (kein if-then-else)
- leichter lesbar

- leichter verifizierbar (vgl. `append`, Kapitel 0, S. 9)

Vergleiche hierzu die prägnante Definition

```
and True  x = x
and False x = False
```

mit der „if-then-else“-Definition

```
and x y = if x==True then y else False
```

oder auch

```
and x y | x==True  = y
        | x==False = False
```

Der musterorientierte Stil kann allerdings auch problematische Fälle haben, deren Bedeutung erklärt werden muss:

- Wie ist die Reihenfolge bei einer Musteranpassung?
  (links-nach-rechts, oben-nach-unten, best-fit?)
- Deklarativ sinnlose Definitionen möglich:

  ```
  f True  = 0
  f False = 1
  f True  = 2
  ```

  Die letzte Gleichung ist aus deklarativer Sicht sinnlos, da die erste Gleichung schon vorschreibt, dass der Wert von “`f True`” immer `0` sein soll.

Spezifikationssprache $\leftrightarrow$ deklarative Programmiersprache

Die **deklarative Programmierung** basiert auf der Definition von Eigenschaften statt der Angabe des genauen Programmablaufs. Hieraus könnte man folgern, dass Spezifikationen identisch mit deklarativen Programmen sind.
Dies ist allerdings nicht der Fall, denn ein Programm ist effektiv ausführbar (z.B. Reduktion von Ausdrücken), während eine Spezifikation eine formale Problembeschreibung ist, die evtl. nicht ausführbar ist.

Da **Programmieren** als Überführung einer Spezifikation in eine ausführbare Form aufgefasst werden kann, kann die Programmierung allerdings durch deklarative Sprachen vereinfacht werden.

Beispiel: **Quadratwurzelberechnung nach Newton**

Wurzelfunktion: $\sqrt{x} = y$, so daß $y \geq 0$ und $y^2 = x$

Dies ist eine *Spezifikation*, die aber nicht effektiv ausführbar ist, denn wie sollen wir einen passenden Wert für y raten?

Ein konstruktives Approximationsverfahren ist das *Newtonsche Iterationsverfahren*:
Gegeben: Schätzung für y
bessere Schätzung: Mittelwert von y und $\frac{x}{y}$

Beispiel:

$x = 2$	$Schätzung:$	$Mittelwert:$
	1	$\frac{1+2}{2} = 1.5$
	1.5	$\frac{1.5+1.3333}{2} = 1.4167$
	1.4167	$\frac{1.4167+1.4118}{2} = 1.4142$

Funktionale Definition: Iteration mit Abbruchbedingung:

```
iter y x = if (ok y x) then y else iter (improve y x) x
```
Abbruch — *Verbesserung*

```
improve y x = (y + x/y) / 2

ok y x = abs (y*y-x) < 0.001
```

Hierbei berechnet `abs` den Absolutbetrag einer Zahl. Damit erhalten wir als Gesamtlösung:

```
wurzel x = iter 1 x

> wurzel 9  ⇝ 3.00009
```

Nachteil dieser Lösung:

- `wurzel`, `iter`, `improve`, `ok` gleichberechtigte (globale) Funktionen
- Argument `x` muss „durchgereicht“ werden
- mögliche Konflikte bei Namen wie `ok`, `improve`

Eine aus Software-technischer Sicht verbesserte Definition erhalten wir durch Benutzung einer **where-Klausel**:

```
wurzel x = iter 1
  where iter y    = if ok y then y else iter (improve y)
        improve y = (y + x/y) / 2
        ok y      = abs (y*y-x) < 0.001
```

Hierbei sind die Deklarationen nach dem Schlüsselwort `where` *lokale Deklarationen*, die nur in der Gleichung für `wurzel` sichtbar sind, d.h. der **Gültigkeitsbereich** der `where`-Deklarationen ist nur die vorhergehende rechte Gleichungsseite (bzw. die Folge von bedingten Gleichungsseiten).

Weitere Form lokaler Deklarationen: **let-Ausdrücke**

`let` <Deklarationen> `in` <Ausdruck>

Die Deklarationen sind hierbei nur im Ausdruck sichtbar.

Beispiel: Definition der Funktion

$$f(x, y) = y(1 - y) + (1 + xy)(1 - y) + xy$$

in Haskell mittels `let`-Deklarationen:

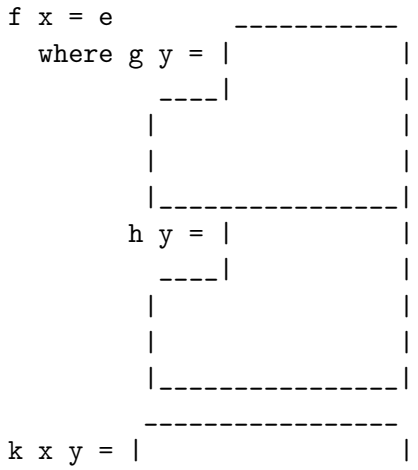
```
f x y = let a = 1-y
            b = x*y
        in  y*a+(1+b)*a+b
```

Wann enden Deklarationen?
Formatfrei: *Klammerung:*

```
let {...;...} in ...

f x = e where {...;...}
```

Formatabhängig: *Abseitsregel* (*offside rule*, *layout rule*):

1. Erstes Symbol nach `let`, `where` (und `of` bei case-Ausdrücken) legt den *linken Rand* des Geltungsbereiches fest.
2. Beginnt neue Zeile rechts vom Rand ⇒ Fortsetzung der vorherigen Zeile
3. Beginnt neue Zeile genau am Rand ⇒ neue Deklaration
4. Beginnt neue Zeile links vom Rand ⇒ Geltungsbereich endet davor, Zeile gehört zum umfassenden Bereich



- gewöhnungsbedürftig, aber:
- erlaubt kompakte, übersichtliche Schreibweise

1.2 Datentypen

Moderne funktionale Sprachen haben *strenges Typkonzept*, d.h.

- alle Werte sind klassifiziert in *Typbereiche* (boolesche Werte, Zeichen, ganze Zahlen, ...)
- jeder zulässige Ausdruck hat einen Typ: falls der Ausdruck zu einem Wert ausgerechnet wird, dann gehört dieser zur Wertmenge, die durch den Typen spezifiziert wird

Wir schreiben

e `::` τ

um auszudrücken, dass der Ausdruck e den Typ τ hat.

Basistypen

Wahrheitswerte: `Bool`
Konstanten: `True`, `False`
Funktionen: `&&` (und) `||` (oder) `not` (nicht)

Ganze Zahlen: `Int` (auch `Integer` für Zahlen beliebiger Größe)
Konstanten: `0, 1, -42, ...`
Funktionen: `+, -, *, /, div, mod, ...`

Gleitkommazahlen: `Float` (auch `Double`)
Konstanten: `0.3 1.5 e-2 ...`

Ganze Zahlen und Gleitkommazahlen sind verschiedene Objekte und es erfolgt bei einem strengen Typsystem keine automatische Konversion zwischen `Int` und `Float`. Haskell hat aber ein mächtiges Typkonzept mit Überladung (Typklassen), das es erlaubt, dass man ganze Zahlkonstanten auch hinschreiben kann, wo Gleitkommazahlen erwartet werden. Allerdings ist ein Ausdruck wie "`div 4 2.0`" unzulässig.

Zeichen: `Char`
Konstanten: `'a' '0' '\n'`

Strukturierte Typen:

Listen (Folgen) von Elementen von Typ `t`: Typnotation: `[t]`

Eine Liste ist

- entweder leer: Konstante `[]`
- oder nicht-leer, d.h. bestehend aus einem (Kopf-) Element `x` und einer Restliste `xs`: `x:xs`

Beispiel: Liste mit Elementen `1,2,3`:

```
1:(2:(3:[])) :: [Int]
```

1. ":" ist rechtsassoziativ: Somit können wir Klammern vermeiden:

   ```
   1 : 2:  3 : []
   ```

2. Direkte Aufzählung aller Elemente: `[1, 2, 3]`
   Somit:

   ```
   [1,2,3]  ==  1:[2,3]  ==  1:2:[3]
   ```

3. Abkürzungen für Listen von Zahlen:

   ```
   [a..b]   == [a, a+1, a+2, ..., b]

   [a..]    == [a, a+1, a+2, ...        -- Unendlich!

   [a,b..c] == [a, a+s, a+2*s, ...,c]   -- mit Schrittweite s=b-a

   [a,b..]  == [a, a+s, a+2*s, ...      -- mit Schrittweite s=b-a
   ```

Operationen auf Listen (viele vordefiniert in der `Prelude`)

`length xs`: Länge der Liste `xs`

```
length []     = 0
length (x:xs) = 1 + length xs
```

"`++`:" Konkatenation zweier Listen (rechtsassoziativ)

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

Strings, Zeichenketten: `String` identisch zu `[Char]`
Konstanten: `"Hallo" == ['H','a','l','l','o']`
Funktion: wie auf Listen

Tupel: $(t_1, t_2, \ldots, t_n)$: Struktur („Record“, kartesisches Produkt) mit Komponenten der Typen $t_1, t_2, \ldots, t_n$
Konstanten: `(1,'a',True) :: (Int,Char,Bool)`

Funktionale Typen: t_1`->`t_2: Funktionen mit Argument von Typ t_1 und Ergebnis von Typ t_2

Beispiel: `square :: Int -> Int`

Konstante:

`\`	x `->`	e
↑	↑	↑
λ(lambda)	*Parameter*	*Rumpf*

äquivalent zu $f\ x = e$, falls f Name dieser Funktion

„λ-Abstraktion“: Definiert „namenlose“ Funktion

Bsp.: `\x -> 3*x+4 :: Int -> Int`

Besonderheit funktionaler Sprachen:
Funktionale Typen haben gleichen Stellenwert wie andere Typen („Funktionen sind Bürger 1. Klasse“, „functions are first class citizens“), d.h. sie können vorkommen in

- Argumenten von Funktionen
- Ergebnissen von Funktionen
- Tupeln
- Listen

⋮

Falls Funktionen in Argumenten oder Ergebnisse anderer Funktionen vorkommen, nennt man diese auch **Funktionen höherer Ordnung**

`polynomdiff:: (Float → Float) → (Float → Float)`

Anwendung (Applikation) einer Funktion f auf Argument x:
mathematisch: $f(x)$
In Haskell: Klammern weglassen: $f\ x$ (Anwendung durch Hintereinanderschreiben)

Funktionen mit mehreren Argumenten:
Funktionale Typen haben nur ein Argument.
Zwei Möglichkeiten zur Erweiterung auf mehrere Argumente:

1. Mehrere Argumente als Tupel:

   ```
   \ (x,y)  → x+y+2 :: (Int,Int)  → Int

   add (x,y) = x+y
   ```

2. „Curryfizierung“ (nach Schönfinkel und Curry)
   - Tupel sind kartesische Produkte: $(A, B) \hat{=} A \times B$
   - Beobachtung: Funktionsräume $(A \times B) \to C$ und $A \to (B \to C)$ sind isomorph
     $\Rightarrow \forall f : (A \times B) \to C$ existiert äquivalente Funktion $f' :: A \to (B \to C)$

   Beispiel:

   ```
   add':: Int  → (Int  → Int)
   add' x = \y  → x+y

   add (x,y) ⇝ x+y
   (add' x) y ⇝ (\y  →x+y) y ⇝ x+y
   ```

   Also: Statt Tupel von Argumenten Hintereinanderanwendung der Argumente
   Interessante Anwendung bei partieller Applikation:

   `(add 1)`: Funktion, die eins addiert $\approx$ Inkrementfunktion
   $\Rightarrow$ `(add 1) 3 = 4`

Viele Anwendungen: generische Funktionen (→ Kapitel 1.4, Funktionen höherer Ordnung)

Daher: Standard in Haskell: curryfizierte Funktionen

$f\ x_1\ x_2 \ldots x_n \hat{=} (\ldots((f\ x_1)\ x_2) \ldots x_n)$ (Applikation linksassoziativ)

$t_1 \to t_2 \to \ldots \to t_n \hat{=} t_1 \to (t_2 \to (\ldots \to t_n) \ldots)$ (Funktionstyp rechtsassoziativ)

Daher:

```
add :: Int  → Int  → Int   =  Int  → (Int  → Int)
add x y = x+y
⋮
add 1 2 ⇝ ((  add 1  ) 2)
            ::Int→Int
```

Achtung: `add` 1 2 $\neq$ `add` $(1, 2)$:
Auf der linken Seite muss `add` den Typ `Int -> Int -> Int` haben
Auf der rechten Seite muss `add` den Typ `(Int, Int) -> Int` haben

Aber: es gibt Umwandlungsfunktionen zwischen Tupeldarstellung und curryfizierter Darstellung (`curry`, `uncurry`).

Benutzerdefinierte (algebraische) Datentypen:

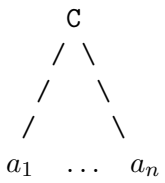
Jedes Objekt eines bestimmten Typs ist immer aufgebaut aus

Konstruktoren: Funktionen mit entsprechendem Ergebnistyp, die nicht reduzierbar sind (frei interpretiert, keine Funktionsdefinition)

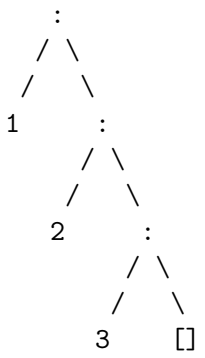
Beispiel:

Typ	**Konstruktoren**
`Bool`	`True False`
`Int`	`0 1 2...-3...`
Listen:	`[] :`

Konstruktion `(C` $a_1 \ldots a_n$`)` entspricht Baumstruktur:



Beispiel: Liste `1:2:3:[]` entspricht Baumstruktur:



Dagegen ist `++` eine Funktion, die Listen (Bäume) verarbeitet: `[1] ++ [2]` $\rightsquigarrow$ `[1,2]`

Konvention in Haskell:

- Konstruktoren beginnen mit Großbuchstaben
- Funktionen beginnen mit Kleinbuchstaben

Damit erfolgt die Definition neuer Datentypen durch Festlegung der Konstruktoren hierfür:
Datendefinition in Haskell:

`data` t `=` C_1 $\mathtt{t}_{11}$ `...` $\mathtt{t}_{1n_1}$ `|` `...` `|` C_k $\mathtt{t}_{k1}$ `...` $\mathtt{t}_{kn_k}$

führt neuen Typ t und Konstruktoren $C_1, \ldots C_k$ mit

$C_i :: t_{11} \to \ldots \to t_{in_i} \to t$

ein.

Beispiel: Neue Listen:

```
data List = Nil | Cons Int List
```

definiert neuen Datentyp List und Konstruktoren

```
Nil  :: List
Cons :: Int  → List  → List
```

Spezialfälle:

Aufzählungstypen:

```
data Color = Red | Yellow | Blue
data Bool = True | False
```

Verbundtypen (Records):

```
data Complex = Complex Float Float

addc :: Complex  → Complex  → Complex
addc (Complex r1 i1) (Complex r2 i2) = Complex (r1+r2) (i1+i2)
```

Rekursive Typen (variante Records): Listen (s.o.)
Binärbäume mit ganzzahligen Blättern:

```
data True = Leaf Int | Node Tree Tree
```

Addiere alle Blätter:

```
addLeaves (Leaf x) = x
addLeaves (Node t1 t2) = (addLeaves t1) + (addLeaves t2)
```

Beachte: Kein Baumdurchlauf, sondern Spezifikation!

1.3 Pattern Matching

Bevorzugter Programmierstil in deklarativen Programmiersprachen:

> Funktionsdefinition durch Muster (pattern) und mehrere Gleichungen

Auswertung von Ausdrücken:

> Auswahl und Anwenden einer passenden Gleichung

Die Auswahl wird dabei auch als "pattern matching" bezeichnet.

Vergleiche: Definition der Konkatenation von Listen:
musterorientiert:

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

Ohne Muster (wie in klassischen Programmiersprachen):

```
append xs ys = if xs == []
               then ys
               else (head xs): (append (tail xs) ys)
```

wobei `head`, `tail` vordefinierte Funktionen auf Listen sind:
`head` liefert das Kopfelement der Liste
`tail` liefert den Rest der Liste
d.h. es gilt immer:

```
head (x:xs) == x
tail (x:xs) == xs
```

bzw.

```
(head xs) : (tail xs) == xs
```

Andere Sprechweise:
":" ist ein **Konstruktor** für den Datentyp Liste
`head`, `tail` sind **Selektoren** für den Datentyp Liste

Somit:
Musterorientierter Stil:

- Schreibe Konstruktoren in Argumente von linken Regelseiten
- haben Wirkung von Selektoren
- linke Seiten: „Prototyp" für den Funktionsaufruf, bei dem diese Regel anwendbar ist
  Beispiel:

```
append [] ys = ys    -- (*)
```

ist anwendbar, falls 1. Argument die Form einer leeren Liste hat und das 2. Argument beliebig ist.

Wann passt ein Muster?

> Ersetze Variablen in Muster ("binde Variablen") so durch andere Ausdrücke, dass das ersetzte Muster syntaktisch identisch zum gegebenen Ausdruck ist. Ersetze in diesem Fall den Ausdruck durch rechte Regelseite (mit den entsprechenden Bindungen!).

Beispiel: Regel `(*)` und Ausdruck

- `append [] [1,2,3]`: Binde `ys` an `[1,2,3]`
  ⇒ linke Seite passt ⇒ Ersetze Ausdruck durch "`[1,2,3]`"
- `append [1] [2,3]`: Muster in `(*)` passt nicht, da Konstruktor `[]` immer verschieden vom Konstruktor "`:`" ist (beachte: `[1] == (:) 1 []`)

Was sind mögliche Muster in Haskell (es gibt aber noch mehr...)?

Muster	Passt, falls
`x`	Variable passt auf jeden Ausdruck und wird an diesen gebunden
`[] True 'a'`	Konstanten passen nur auf gleichen Wert
`(x:xs) (Node` $t_1 t_n$`)`	Konstruktoren passen nur auf gleichen Konstruktor. Zusätzlich müssen die Argumente jeweils passen
(t_1, t_2, t_3)	Tupel passt mit Tupel gleicher Stelligkeit (Argumentzahl), zusätzlich müssen die Argumente jeweils passen
`(x+`k`)` (k=natürliche Zahl)	Passt auf ganze Zahlen $\geq k$, wobei dann `x` an die Zahl minus k gebunden wird Beispiel: `sub3 (n+3) = n`
`v@pat`	Passt, falls Muster `pat` "passt". Zusätzlich wird `v` an gesamten Wert gebunden Beispiel: Statt: `fac (n+1) = (n+1) * fac n` auch möglich: `fac v@(n+1) = v * fac n` Effekt: Vermeidung des Ausrechnens von `(n+1)` in der rechten Seite ⇒ Effizienzsteigerung
`_`	Joker, "wildcard pattern": Passt auf jeden Ausdruck, keine Bindung

Problem: Muster mit mehrfachem Variablenvorkommen:

```
equ x x = True
```

Woran wird Variable `x` gebunden?
⇒ Solche Muster sind unzulässig!

In Mustern kommt jede Variable höchstens einmal vor.

Ausnahme: `_` : Jedes Vorkommen steht für eine andere Variable

Problem: Anwendbare Ausdrücke an Positionen, wo man den Wert wissen muss

Beispiel:

```
fac 0 = 1
```

Aufruf: `fac (0+0)`
`(0+0)` → Passt erst nach Auswertung zu `0`

⇒ Falls beim pattern matching der Wert relevant ist (d.h. bei allen Mustern außer Variablen), wird der entsprechende aktuelle Parameter ausgewertet.

Die Form der Muster beeinflusst das Auswertungsverhalten und damit auch das Terminationsverhalten!

Beispiel: Betrachte folgende Definition der Konjunktion: ("Wahrheitstafel"):

```
and   False   False   =  False
and   False   True    =  False
and   True    False   =  False
and   True    True    =  True
```

Effekt der Muster: bei `(and` t_1 t_2`)` müssen beide Argumente ausgewertet werden, um eine Regel anzuwenden.
Betrachte nichtterminierende Funktion:

```
loop :: Bool
loop = loop   -- Auswertung von loop terminiert nicht
```

Dann: `(and False loop)`
terminiert nicht

Verbesserte Definition durch Zusammenfassen von Fällen:

```
and1 False x = False
and1 True  x = x
```

Effekt: 2. Argument muss nicht ausgewertet werden:
⇒ `(and1 False loop)` $\rightsquigarrow$ `False`

Weiteres Problem: Reihenfolge der Abarbeitung von Mustern:

Logisches Oder ("parallel or")

```
or True    x    = True       (1)
or  x    True   = True       (2)
or False False  = False      (3)
```

Betrachte: `(or loop True)`
Regel(2) passt (mit `x` $\mapsto$ `loop`): `True`
Aber: Regel (1) könnte auch passen: Auswertung des 1. Argumentes $\rightsquigarrow$ Endlosschleife

Dies zeigt, dass die Musterabarbeitung in einer Programmiersprache genau festgelegt werden muss. In Haskell ist diese informell wie folgt definiert:

> Alle Regeln werden von oben nach unten im Programm ausprobiert. Bei jeder Regel werden die Muster links nach rechts abgearbeitet. Hier wird ein Argument ausgewertet, falls ein Muster in einer Regel dies verlangt.

Nachfolgend wollen wir diese Strategie präzisieren. Hier verwenden wir `case`-Ausdrücke, die die folgende Form haben (Haskell erlaubt auch allgemeinere Formen):

`case e of`

$$\begin{array}{l} C_1\ x_{11} \ldots x_{1n_1} \to e_1 \\ \quad\vdots \qquad\qquad\quad \vdots \\ \quad\vdots \qquad\qquad\quad \vdots \\ C_k\ x_{k1} \ldots x_{kn_k} \to e_k \end{array}$$

wobei $C_1, \ldots, C_k$ Konstruktoren des Datentyps von e sind.

Beispiel: Mittels eine `case`-Ausdrucks kann die obige `append`-Operation auch wie folgt definiert werden:

```
append xs ys = case xs of
                 []    → ys
                 x:zs  → x : append zs ys
```

Bedeutung des `case`-Ausdrucks:

1. Werte e aus (bis ein Konstruktor oben steht)
2. Falls die Auswertung $C_i\ a_1 \ldots a_{n_i}$ ergibt (sonst: Fehler), binde x_{ij} an a_j ($j = 1, \ldots, n_i$) und ersetze den gesamten `case`-Ausdruck durch e_i

Obiges Beispiel zeigt: Pattern matching in case-Ausdrücken übersetzbar

Im Folgenden: Definition der Semantik von Pattern matching durch Übersetzung in `case`-Ausdrücke.

Gegeben: Funktionsdefinition

$f\ p_{11} \ldots p_{1n}$ `=` e_1
$\vdots$
$f\ p_{m1} \ldots p_{mn}$ `=` e_m

Wir übersetzen diese Definition in `case`-Ausdrücke mittels einer Transformation `match`, die wir nachfolgend genauer definieren:

`f` $x_1 \ldots x_n$ `=`
  ⟦`match [`x_1`, ..., `x_n`] [([`p_{11}`, ..., `p_{1n}`], `e_1`),`
  $\vdots$
  `([`p_{m1}`, ..., `p_{mn}`], `e_m`)]`
  `ERROR`⟧

Die Klammern ⟦ ... ⟧ bedeuten, dass der darin enthaltene Ausdruck (zur Übersetzungszeit) zu einem Programmausdruck ausgewertet wird.

Um die Funktion der Transformation `match` besser zu verstehen, betrachten wir die allgemeine Form

```
match xs eqs E
```

Hier ist

`xs`: Liste von Argumentvariablen

`eqs`: Liste von Pattern-Rumpf-Paaren

`E`: Fehlerfall, falls kein Pattern aus `eqs` passt

Wir definieren `match` durch vier Transformationsregeln:

1. **Alle Muster fangen mit einer Variablen an:**

   ⟦`match (x:xs) [(`$v_1 : ps_1$`, `e_1`),`
   $\vdots$
   `(`$v_m : ps_m$`, `e_m`)] E`⟧
   `=` ⟦`match xs [(`ps_1`, `$e_1[v_1 \mapsto \texttt{x}]$`),`
   $\vdots$
   `(`ps_m`, `$e_m[v_m \mapsto \texttt{x}]$`)] E`⟧

   Hierbei steht $e[v \mapsto t]$ für die Ersetzung aller Vorkommen der Variablen v in e durch t.

   Somit werden bei Variablenmustern keine Argumente ausgewertet.

   Anmerkung: diese Transformationsregel ist auch anwendbar falls $m = 0$ (leere Gleichungsliste).

2. **Alle Muster fangen mit einem Konstruktor an:**
Dann Argument auswerten + Fallunterscheidung über Konstruktoren:

$$[\![\texttt{match (x:xs) eqs E}]\!] \texttt{ -- wobei eqs}\neq\texttt{[]}$$
$$= [\![\texttt{match (x:xs) (eqs}_1 \texttt{ ++ } \cdots \texttt{ ++ eqs}_k\texttt{) E}]\!]$$

Hierbei ist $(\texttt{eqs}_1 \texttt{ ++ } \cdots \texttt{ ++ eqs}_k)$ eine Gruppierung von `eqs`, so dass alle Muster in $\texttt{eqs}_i$ mit dem Konstruktor C_i beginnen, wobei $C_1, \ldots, C_k$ alle Konstruktoren vom Typ von `x` sind, d.h.

$$\begin{array}{lll} \texttt{eqs}_i & = & \texttt{[}((C_i\ \ p_{i,1,1} \ldots p_{i,1,n_i}) : ps_{i,1},\ \ e_{i,1})\texttt{,} \\ & & \quad\vdots \\ & & \ ((C_i\ \ p_{i,m_i,1} \ldots p_{i,m_i,n_i}) : ps_{i,m_i},\ \ e_{i,m_i})\texttt{]} \end{array}$$

Beachte: falls in `eqs` der Konstruktor C_j nicht vorkommt, dann ist $\texttt{eqs}_j = \texttt{[]}$. Außerdem haben in jedem $\texttt{eqs}_i$ die Regeln die gleiche Reihenfolge wie in `eqs`.

Nun transformieren wir weiter:

$$\begin{array}{l} [\![\texttt{match (x:xs) (eqs}_1 \texttt{ ++ } \cdots \texttt{ ++ eqs}_k\texttt{) E}]\!] = \\ \quad \texttt{case x of} \\ \quad\quad C_1\ \ x_{1,1} \ldots x_{1,n_1} \ \ \rightarrow [\![\texttt{match ([}x_{1,1}, \ldots, x_{1,n_1}\texttt{]++xs) } eqs'_1 \texttt{ E}]\!] \\ \quad\quad\quad \vdots \\ \quad\quad C_k\ \ x_{k,1} \ldots x_{k,n_k} \ \ \rightarrow [\![\texttt{match ([}x_{k,1}, \ldots, x_{k,n_k}\texttt{]++xs) } eqs'_k \texttt{ E}]\!] \end{array}$$

wobei

$$\begin{array}{lll} eqs'_i & = & \texttt{[(}[p_{i,1,1}, \ldots, p_{i,1,n_i}] \texttt{ ++ } ps_{i,1}\texttt{,}\ \ e_{i,1}\texttt{),} \\ & & \quad\vdots \\ & & \ \texttt{(}[p_{i,m_i,1}, \ldots, p_{i,m_i,n_i}] \texttt{ ++ } ps_{i,m_i}\texttt{,}\ \ e_{i,m_i}\texttt{)]} \end{array}$$

und $x_{i,j}$ neue Variablen sind.

Beispiel: Übersetzung von `append`:

```
append x1 x2
 = ⟦match [x1, x2] [(([], ys], ys),
                    ([x:xs, ys], x: append xs ys)] ERROR⟧
 = case x1 of
     []     →  ⟦match [x2] [([ys], ys)] ERROR⟧
     x3:x4  →  ⟦match [x3, x4, x2] [([x,xs,ys], x: append xs ys)] ERROR⟧
```

Nun wenden wir die 1. Transformation auf den ersten `case`-Zweig an:

```
= case x1 of
    []     → ⟦match [] [([], x2)] ERROR⟧
    x3:x4  → ⟦match [x3, x4, x2] [([x,xs,ys], x: append xs ys)] ERROR⟧

= case x1 of
    []     → x2
    x3:x4  → ⟦match [x3, x4, x2] [([x,xs,ys], x: append xs ys)] ERROR⟧
```

Der letzte Transformationschritt ist sinnvoll, da die Musterliste in

```
⟦match [] [([], x2)] ERROR⟧
```

leer ist und somit das Pattern Matching erfolgreich war. Genau hierfür benötigen wir die nachfolgende Transformationsregel.

3. **Musterliste ist leer:**
   Hier können wir zwei Fälle unterscheiden:
   a) Genau eine Regel ist anwendbar:

      ```
      ⟦match [] [([] e)] E⟧ = e
      ```

   b) Keine Regel ist anwendbar: Benutze nun die Fehleralternative:

      ```
      ⟦match [] [] E⟧ = ⟦E⟧
      ```

   Anmerkung: Theoretisch kann evtl. auch mehr als eine Regel übrigbleiben, z.B. bei

   ```
   f True  = 0
   f False = 1
   f True  = 2
   ```

   In diesem Fall kann der Übersetzer z.B. eine Fehlermeldung machen oder einfach die erste Alternative wählen.

4. **Muster fangen mit Konstruktoren als auch Variablen an:**
   Gruppiere diese Fälle:

   ```
   ⟦match xs eqs E⟧ =
     ⟦match xs eqs_1 (match xs eqs_2 (...(match xs eqs_n E) ...))⟧
   ```

   wobei

   ```
   eqs == eqs_1 ++ eqs_2 ++ ...++ eqs_n
   ```

   und in jedem $\texttt{eqs}_i$ $(\neq\texttt{[]})$ beginnen alle Muster entweder mit Variablen oder mit Konstruktoren; falls in $\texttt{eqs}_i$ alle Muster mit Variablen anfangen, dann fangen in

`eqs`$_{i+1}$ alle Muster mit Konstruktoren an (oder umgekehrt)

Dieser Algorithmus hat folgende Eigenschaften:

1. **Vollständigkeit**: Jede Funktionsdefinition wird in `case`-Ausdrücke übersetzt (dies ist klar wegen der vollständigen Fallunterscheidung aller Transformationsregeln).
2. **Terminierung**: Die Anwendung der Transformationen ist endlich.
   Definiere hierzu:
   - Größe eines Musters: Anzahl der Symbole in diesem
   - Größe einer Musterliste: Summe der Größen der einzelnen Muster

   Dann gilt: jeder `match`-Aufruf wird durch Aufrufe mit kleinerer Musterlistengröße in einer Transformation ersetzt, woraus die Terminierung des Verfahrens folgt.

Beispiel: Übersetzung der logischen Oder-Operation:

```
or  True   x     = True    (1)
or  x      True  = True    (2)
or  False  False = False   (3)
```

Im ersten Schritt wird diese Operation wie folgt übersetzt:

```
or x y =   ⟦ match [x,y] [([True , x    ], True),
                          ([x    , True ], True),
                          ([False, False], False)] ERROR ⟧
```

Die Anwendung der 4. Transformation auf den `match`-Ausdruck ergibt:

```
match [x,y] [([True, x], True)]
  (match [x,y] [([x, True], True)]
    (match[x,y] [([False, False], False)] ERROR))
```

Dieser Ausdruck wird wie folgt weiter übersetzt, wobei die Art der einzelnen Transformationsschritte links notiert ist:

```
2          case x of
1,3          True   → True
1,2          False  → case y of
3                       True  → True
2                       False → case x of
3                                  True   → ERROR
2                                  False  → case y of
3                                             True   → ERROR
3                                             False  → False
```

Nun ist klar, wie `(or loop True)` abgearbeitet wird: zunächst erfolgt eine Fallunterscheidung über das 1. Argument, was zu einer Endlosschleife führt.

Allerdings gibt es keine Endlosschleife, wenn die Regeln `(1)` und `(2)` in der Reihenfolge vertauscht werden!

Ursache: Regeln `(1)` und `(2)` "überlappen", d.h. für das Muster `(or True True)` sind beide Regeln anwendbar

Somit gilt:

> **Bei Funktionen mit überlappenden Mustern hat die Regelreihenfolge Einfluss auf den Erfolg des Pattern Matching.**

Als Konsequenz gilt die Empfehlung:

> **Vermeide überlappende linke Regelseiten!**

Ist dies allerdings ausreichend für Reihenfolgeunabhängigkeit?
Nein! Betrachte hierzu das folgende Beispiel:

```
diag x      True   False  =  1
diag False  x      True   =  2
diag True   False  x      =  3
```

Diese Gleichungen sind nicht überlappend. Betrachten wir den folgenden Ausdruck:

```
diag loop True False
```

Dieser Ausdruck wird zu `1` ausgewertet. Wenn allerdings die Gleichungen in umgekehrter Reihenfolge umsortiert werden, führt die Auswertung dieses Ausdrucks zu einer Endlosschleife!

Die Ursache liegt in einem **prinzipiellem Problem:**

> Es existiert nicht für jede Funktionsdefinition eine sequentielle Auswertungsstrategie, die immer einen Wert berechnet.
> (vgl. dazu [Huet/Lévy 79, Huet/Lévy 91]).

Die Ursache für diese Probleme liegt in der Tranformation 4, bei der Auswertung und Nichtauswertung von Argumenten gemischt ist. Daher definieren wir die folgende Klasse von Funktionen: ⇒

> Eine Funktionsdefinition heißt **uniform**, falls die Transformation 4 bei der Erzeugung von `case`-Ausdrücken nicht benötigt wird.

Intuitiv: Keine Vermischung von Variablen und Konstruktoren an gleichen Positionen.

Satz 1.1 *Bei uniformen Definitionen hat die Reihenfolge der Regeln keinen Einfluss auf das Ergebnis.*

Beispiel:

```
xor False x     = x
xor True  False = True
xor True  True  = False
```

ist uniform (trotz Vermischung im 2. Argument) wegen der „links-nach-rechts“-Strategie beim Pattern-Matching.

Dagegen ist

```
xor' x     False = x
xor' False True  = True
xor' True  True  = False
```

nicht uniform. Es wäre aber uniform bei Pattern matching von rechts nach links.

Es sind auch andere Pattern-Matching-Strategien denkbar, aber viele funktionale Sprachen basieren auf links→rechts Pattern Matching.

Ausweg: Nicht-uniforme Definitionen verbieten? Manchmal sind sie aber ganz praktisch:

Beispiel: Umkehrung einer 2-elementigen Liste (und nur diese!):

```
rev2 [x,y] = [y,x]
rev2 xs    = xs
```

Als uniforme Definition:

```
rev2  []         = []
rev2  [x]        = [x]
rev2  [x,y]      = [y,x]
rev2  (x:y:z:xs) = x:y:z:xs
```

Andere Alternativen: nicht-sequentielle Auswertung, d.h. statt leftmost outermost könnte man auch eine „parallel outermost“-Strategie verwenden. Diese wäre zwar vollständig (d.h. läuft nicht in Endlosschleifen, obwohl Werte existieren), sie ist aber aufwändig zu implementieren.

1.4 Funktionen höherer Ordnung

Charakteristische Eigenschaft funktionaler Programmiersprachen:

> Funktionen sind „Bürger 1. Klasse“, d.h. sie können überall da auftreten, wo „Werte“ verlangt werden, also als Parameter oder Ergebnis anderer Funktionen (in diesem Fall heißen diese „Funktionen höherer Ordnung“) oder in Datenstrukturen.

Anwendung:

- generische Funktionen

- Programmschemata

$\Rightarrow$ mehr Wiederverwendbarkeit und Modularität

Im Prinzip ist dies ein bekanntes Konzept aus der Mathematik:

Beispiel: Die Ableitung einer Funktion nach einer Veränderlichen kann aufgefasst werden als Funktion

$$\texttt{deriv :: } \underbrace{\texttt{(Float -> Float)}}_{Eingabefunktion} \texttt{ -> } \underbrace{\texttt{(Float -> Float)}}_{Ausgabefunktion}$$

Numerische Berechnung der 1. Ableitung:

$$f'x = \lim_{dx \to 0} \frac{f(x+dx) - f(x)}{dx}$$

Implementierung dieser Funktion als Näherung (d.h. wähle kleines dx):

```
deriv f = f'
      where f' x = (f (x+dx) - f x) / dx
            dx = 0.00001
```

Anwendung:

$\underbrace{\texttt{deriv sin 0.0}}_{\texttt{cos}} \rightsquigarrow 1.0$

$\underbrace{\texttt{deriv (\x -> x*x)}}_{\texttt{2*x}} 1.0 \rightsquigarrow 2.00272$

Wichtige Anwendung von Funktionen höherer Ordnung:
Definition von *Programmierschemata*
Beispiel:

- Quadrieren aller Elemente einer Zahlenliste:

  ```
  sqlist []     = []
  sqlist [x:xs] = (square x): sqlist xs
  ```

  Damit erhalten wir z.B. folgendes Ergebnis:

  ```
  sqlist [1,2,3,4]  ⇝  [1,4,9,16]
  ```

- Chiffrieren von Strings durch zyklische Verschiebung um 1 Zeichen:

  ```
  code c | c=='Z'    = 'A'
         | c=='z'    = 'a'
         | otherwise = chr (ord c + 1)
  ```

  Hierbei ist:

```
chr :: Int  → Char
ord :: Char  → Int
```

Anwendung auf Strings:

```
codelist  []    = []
codelist (c:cs) = (code c): codelist cs
codelist 'ABzXYZ' → 'BCaYZA'
```

Anmerkung:

- `sqlist` und `codelist` haben ähnliche Struktur
- Unterschied: „`code`“ statt „`square`“

Verallgemeinerung durch eine Funktion mit Funktionsparameter:

```
map :: (a  → b)  → [a]  → [b]
map f []     = []
map f (x:xs) = f x: map f xs
```

Bei der Angabe des Typs von `map` sind `a` und `b` Typvariablen, die durch beliebige Typen, wie z.B. `Int` oder `Char`, ersetzbar sind (dies wird genauer im Kapitel 1.5 erläutert).

Nun erhalten wir obige Funktionen als Spezialisierung von `map`:

```
sqlist xs   = map square xs
codelist cs = map code cs
```

Der Vorteil ist also, dass `map` universell einsetzbar ist:

- Andere Codierungsfunktion für Zeichen, z.B.

```
rot13:: Char  → Char           -- verschiebe um 13 Zeichen
rot13list cs = map rot 13 cs
```

- Inkrementierung einer Zahlenliste um 1:

```
inclist xs = map (\x  → x+1) xs
```

Auffallend: Bei allen Definitionen ist das letzte Argument (`xs`, `cs`) identisch. Diese Beobachtung können wir ausnutzen, indem wir „Funktionsgleichungen“ aufschreiben, d.h. wir definieren, welche funktionalen Werte zueinander gleich sind. In einer Sprache mit Funktionen höherer Ordnung, d.h. wo Funktionen Werte sind, ist dies erlaubt:

```
sqlist   = map square
codelist = map code
inclist  = map (\x  → x+1)
```

Hier nennt man die Anwendung von `map` auf nur ein Argument auch eine *partielle Applikation*, weil für eine vollständige Anwendung noch Argumente fehlen.

Partielle Applikationen sind auch nützlich in anderen Anwendungen:

```
add x y = x+y
inc = add 1       -- partielle Anwendung
```

Somit ist `inc` keine Konstante, sondern eine Funktion vom Typ `Int → Int`

Voraussetzung für die partielle Anwendung ist allerdings, dass die angewendete Funktion einen curryfizierten Typ hat, d.h. der Typ hat die Form

$$t_1 \to t_2 \to \ldots \to t_n$$

(vgl. Kapitel 1.2)

Zu Erinnerung:
$f\ x$ steht immer für die (partielle) Anwendung von f auf ein Argument, d.h. falls

$$f \texttt{ :: } t_1 \to t_2 \to \cdots \to t_n$$

und

$$x \texttt{ :: } t_1$$

dann ist $f\ x :: t_2 \to \cdots \to t_n$.

Sonderfall: Bei Infixoperationen (`+`, `-`, `*`, `/`, `++`, ...) gibt es zwei mögliche partielle Anwendungen:
Präfixapplikation: `(2-)` $\hat{=}$ `\x → 2 - x`
Postfixapplikation: `(-2)` $\hat{=}$ `\x → x - 2`

Folgende Ausdrücke sind daher äquivalent:

```
a/b   ==   (/)  a b   ==   (a/)b   ==   (/b) a
```

In Kapitel 1.2 hatten wir erwähnt, dass die Funktionsräume `a → b → c` und `(a,b) → c` isomorph sind. Tatsächlich ist es möglich, curryifizierte und nicht-curryfizierte Funktionen ineinander zu übersetzen:

„Curryfizieren“:

```
curry :: ((a,b) → c) → (a → b → c)
curry f x y = f (x,y)

-- Alternative Definition:
curry f = \x → \y → f(x,y)
```

Umkehrung:

```
uncurry :: (a → b → c) → ((a,b) → c)
uncurry f (x,y) = f x y
```

Es gilt:

```
uncurry (curry f) = f
curry (uncurry g) = g
```

Allgemeiner formuliert:

```
uncurry . curry = id
curry . uncurry = id
```

wobei "`id`" die Identitätsfunktion

```
id x = x
```

und "`.`" die Funktionskomposition ist:

```
f . g = \x → f (g x)
```

Beispiel: Falls

```
(+) :: Int → Int → Int
```

dann ist

```
(uncurry (+)) :: (Int, Int) → Int
```

Diese Übersetzungen kann man z.B. wie folgt anwenden:

Gegeben sei eine Liste von Zahlenpaaren, z.B. `[(2,3),(5,6)]`.

Aufgabe: Addiere aller Zahlen in der Liste von Zahlenpaaren:

```
(sum . map (uncurry (+))) [(2,3), (5,6)]   ⇝ 16
```

Hierbei verwenden wir die vordefinierte Operation `sum` zur Aufsummierung aller Elemente einer Zahlenliste:

```
sum []     = 0
sum (x:xs) = x + sum xs
```

Multiplizieren aller Zahlen in der Liste von Zahlenpaaren:

```
(prod . map (uncurry (*))) [(2,3),(5,6)]    ⇝ 180
```

wobei `prod` alle Elemente einer Zahlenliste multipliziert:

```
prod []     = 1
prod (x:xs) = x * prod xs
```

Auffällig ist, dass die Definitionen von `sum` und `prod` einem gleichen Schema folgen. Dieses Schema können wir durch ein einziges Schema verallgemeinern, das über eine binäre Operation, ein (neutrales) Element und einer Liste von Werten parametrisiert ist und alle Elemente dieser Liste mit der binären Operation und dem Element verknüpft:

```
foldr :: (a → b → b) → b → [a] → b
foldr f e []     = e
foldr f e (x:xs) = f x (foldr f e xs)
```

Damit werden `sum` und `prod` zu Spezialfällen dieses allgemeinen Schemas:

```
sum  = foldr (+) 0
prod = foldr (*) 1
```

Einfache weitere Anwendungen:

```
and    = foldr (&&) True   -- Konjunktion boolescher Werte
concat = foldr (++) []     -- Konkatenation von Listen von Listen
```

Hier ist ein wichtiges Vorgehen bei der (funktionalen) Programmierung ersichtlich:

1. Suche nach allgemeinen Programmschemata (häufig: Rekursionsschema bei rekursiven Datenstrukturen)
2. Realisiere Schema durch Funktion höherer Ordnung

`map` und `foldr` sind Rekursionsschemata für Listen

Ein weiteres Schema auf Listen ist:
Filtere alle Elemente, die ein gegebenes Prädikat erfüllen.

```
filter :: (a → Bool) → [a] → [a]
filter p []     = []
filter p (x:xs) | p x       = x : filter p xs
                | otherwise = filter p xs
```

Anwendung: Umwandlung einer Liste in eine Menge (d.h. Entfernen aller Duplikate):

```
rmdups [] = []
rmdups (x:xs) = x: rmdups (filter (x /=) xs)
```

Anwendung: Sortieren von Zahlenlisten mittels Quicksort:

```
qsort [] = []
qsort (x:xs) = qsort (filter (<= x) xs) ++
               [x] ++
               qsort (filter (> x) xs)
```

Hierdurch erhalten wir z.B. folgendes Ergebnis:

```
qsort [3,1,5,3] ⇝ [1,3,4,5]
```

Schema: Längster Präfix, dessen Elemente ein Prädikat erfüllen

```
takeWhile :: (a → Bool) → [a] → [a]
takeWhile p []     = []
takeWhile p (x:xs) | p x       = x : takeWhile p xs
                   | otherwise = []
```

Dieses Schema entspricht einer **Kontrollstruktur**!

Hier ist ein weiterer Vorteil funktionaler Sprachen ersichtlich: Wir können eigene Kontrollstrukturen durch Funktionen höherer Ordnung (Schemata) definieren.

Beispiel: Eine `until-do`-Schleife besteht aus:

- Abbruchbedingung (Prädikat vom Typ `a -> Bool`)
- Rumpf: Transformation auf Werte (Funktion vom Typ `a -> a`)
- Anfangswert

Beispiel:

```
x:=1                        Anfangswert: 1
until x > 100               Abbruch: p x = x>100
do    x:= 2*x               Transformation: f x = 2*x
```

Implementierung dieser Kontrollstruktur in einer funktionalen Sprache:

```
until:: (a → Bool) → (a → a) → a → a
until p f x | p x       = x
            | otherwise = until p f (f x)
```

wobei die Parameter die folgende Bedeutung haben:
`p`: Abbruch
`f`: Transformation
`x`: Anfangswert

Obige Schleife mit diesem Schema:

```
until (>100) (2*) 1    ⇝ 128
```

Beachte:

- keine Spracherweiterung notwendig!
- auch andere Kontrollstrukturen möglich
- Charakteristik einer guten Programmiersprache:

- einfaches Grundkonzept
- Techniken zur flexiblen Erweiterung
- Bibliotheken

Negativbeispiel: PL/I (komplexe Sprache, enthält viele Konstrukte)

Anwendung: **Newtonsches Verfahren zur Nullstellenberechnung**
Gegeben: Funktion f, Schätzwert x für Nullstelle
Methode:

$$x - \frac{f(x)}{f'(x)}$$

ist eine bessere Näherung

Anwendung: Berechne die Quadratwurzel von a durch Anwendung des Verfahrens auf $f(x) = x^2 - a$

Da dies ein Iterationsverfahren ist, können wir es durch eine Schleife realisieren (das erste Argument ist die Funktion, das zweite Argument ist der Schätzwert):

```
newton:: (Float → Float) → Float → Float
newton f = until ok improve
   where ok x = abs (f x) < 0.001
         improve x = x - f x / deriv f x
```

Berechnung der Quadratwurzel:

```
sqrt x = newton f x  where f y = y*y - x

> sqrt 2.0   ⇝ 1.41421
```

Berechnung der Kubikwurzel:

```
cubrt x = newton f x  where f y = y*y*y - x

> cubrt 27.0 ⇝ 3.00001
```

Funktionen als Datenstrukturen

Was ist eine Datenstruktur?
Ein Objekt mit Operationen zur

- Konstruktion (z.B. `[]`, "`:`" bei Listen)
- Selektion (z.B. `head`, `tail` bei Listen)
- Verknüpfung (z.B. "`++`" bei Listen (optional)

Wichtig ist also nur die Funktionalität (Schnittstelle), aber nicht die Repräsentation der Daten (dies ist die generelle Idee des Konzeptes eines "abstrakter Datentyps").

Daher: Eine Datenstruktur ist im Prinzip ein Satz von Funktionen.

Beispiel: Felder mit Elementen vom Typ `a`:

Konstruktion von Feldern:

```
emptyarray :: Array a   -- leeres Feld
putidx :: Array a → Int → a → Array a
```

Hierbei liefert "`putidx ar i v`" ein neues Feld, wobei an der Indexposition `i` der Wert `v` steht und alle anderen Positionen unverändert sind (beachte: das Feld `ar` selbst kann nicht verändert werden wegen der Seiteneffektfreiheit funktionaler Sprachen!).

Selektion in Feldern:

```
getidx :: Array a → Int → a
```

Hierbei liefert "`getidx ar i`" das Element an der Position `i` im Feld `ar`.

Mehr muss ein Anwender von Feldern nicht wissen!

Wir wollen nun Felder implementieren, d.h. die obigen Funktionen praktisch umsetzen. Hier gibt es verschiedene Möglichkeiten, wobei die einfachste darin besteht, die formale Definition von Feldern direkt umzusetzen. Formal ist ein Feld eine Abbildung von Indizes (vom Typ `Int`) auf Werte, d.h. ein Feld entspricht einer Funktion vom Typ `Int → a`. In Haskell können wir dies einfach durch ein Typsynonym definieren:

```
type Array a = Int → a
```

Dann ist Implementierung der Schnittstelle einfach:

```
emptyarray i = error "subscript out of range"

getidx ar i = ar i

puitdx ar i v = ar'   -- neues Feld, d.h. neue Funktion:
    where ar' j | i==j      = v
                | otherwise = ar j
```

Somit:

```
getidx (putidx emptyarray 2 'b') 2 ⇝ 'b'

getidx (putidx emptyarray 2 'b') 1 ⇝ error ...
```

Somit können wir also auch *Datenstrukturen als Funktionen* implementieren.

Vorteil: konzeptuelle Klarheit (≈ Spezifikation)

Nachteil: Die Zugriffszeit ist abhängig von der Anzahl der `putidx`-Operationen. Allerdings könnte man dies durch eine andere Datenstruktur verbessern (insbesondere, falls das "alte" Feld nicht mehr benötigt wird).

1.5 Typsystem und Typinferenz

Haskell (auch Standard ML oder Java5) ist eine *streng getypte Sprache mit einem polymorphen Typsystem*. Die Bedeutung dieser Begriffe wollen wir zunächst informell erläutern, bevor wir dann später diese genau definieren.

Streng getypte Sprache:

- Jedes Objekt (Wert, Funktion) hat einen *Typ*
- Typ $\approx$ Menge möglicher Werte:

  `Int` $\approx$ Menge der ganzen Zahlen bzw. endliche Teilmenge,

  `Bool` $\approx \{$`True`, `False`$\}$

  `Int` $\rightarrow$ `Int` $\approx$ Menge aller Funktionen, die ganze Zahlen auf ganze Zahlen abbilden
- "Objekt hat Typ τ" $\approx$ Objekt gehört zur Wertemenge des Typs τ
- *Typfehler:* Anwendung einer Funktion auf Werte, die nicht ihrem Argumenttyp entsprechen
  Beispiel: `3 + 'a'`
  Aufruf von `+` ohne Typprüfung der Argumente $\Rightarrow$ unkontrolliertes Verhalten (Systemabsturz, Sicherheitslücken)
- Eigenschaft streng getypter Sprachen:
  Typfehler können nicht auftreten ("well-typed programs do not go wrong", [Milner 78])
  Aus diesem Grund wird der Ausdruck

  ```
  3 + 'a'
  ```

  durch den Compiler zurückgewiesen, ebenso wie der Ausdruck

  ```
  foldr (+) 0 [1, 3, 'a', 5]
  ```

  Beachte: der Typfehler im ersten Ausdruck ist „unmittelbar" klar, wohingegen der Typfehler im zweiten Ausdruck etwas mehr Überlegung benötigt.

Schwach getypte Sprachen (Scheme, Smalltalk, JavaScript, PHP,...):

- Objekte haben Typ (werden zur Laufzeit mitgeführt)
- Funktionen prüfen zur Laufzeit, ob ihre Argumente typkorrekt sind

Vorteile stark getypter Sprachen:

- Programmiersicherheit: Typfehler treten nicht auf
- Programmiereffizienz: Compiler meldet Typfehler, bevor das Programm ausgeführt wird (Compiler prüft „Spezifikation“)
- Laufzeiteffizienz: Typprüfung zur Laufzeit unnötig
- Programmdokumentation: Typangaben können als (automatisch überprüfbare!) Teilspezifikation angesehen werden

Nachteile:

- strenge Typsysteme schränken die Flexibilität ein, um eine automatische Prüfung zu ermöglichen
- nicht jedes Programm ohne Laufzeittypfehler ist zulässig, z.B. ist

  ```
  takeWhile (<3) [1, 2, 3, 'a']
  ```

  unzulässig, obwohl kein Laufzeittypfehler auftreten würde.

Wichtige Ziele bei der Entwicklung von Typsystemen:

- Sicherheit
- Flexibilität (möglichst wenig einschränken)
- Komfort (möglichst wenig explizit spezifizieren)

In funktionalen Sprachen wird dies erreicht durch

Polymorphismus: Objekte (Funktionen) können mehrere Typen haben

Typinferenz: nicht alle Typen müssen deklariert werden, sondern Typen von Variablen und Funktionen werden inferiert

Nachfolgend werden wir beide Konzepte genauer erläutern.

1.5.1 Typpolymorphismus

Wie oben erläutert, bedeutet **Polymorphismus** in Programmiersprachen, dass Objekte, typischerweise Funktionen, mehrere Typen haben bzw. auf Objekte unterschiedlicher Typen angewendet werden können.

Generell unterscheiden wir zwei Arten von Polymorphismus.

Ad-hoc Polymorphismus: Funktionen, die mehrere Typen haben, verhalten sich auf Typen unterschiedlich
Beispiel: **Overloading**, **überladene Bezeichner**:
hier verwendet man einen Bezeichner für unterschiedliche Funktionen
Java: “+” steht für

- Addition auf ganzen Zahlen
- Addition auf Gleitkommazahlen
- Stringkonkatenation

Parametrischer Polymorphismus: Funktionen haben gleiches Verhalten auf allen ihren zulässigen Typen

Beispiel: Berechnung der Länge einer Liste:

```
length []     = 0
length (x:xs) = 1 + length xs
```

Die Berechnung ist unabhängig vom Typ der Elemente
⇒ Typ von `length`:

```
length :: [a]  → Int
```

Hierbei ist `a` eine **Typvariable**, die durch jeden anderen Typ ersetzbar ist
⇒ `length` anwendbar auf `[Int]`, `[Float]`, `[[Int]]`, `[(Int, Float)]`, ...
z.B. ist der Ausdruck

```
length [0,1] + length ['a', 'b', 'c']
```

zulässig!
Im Gegensatz zum ad-hoc Polymorphismus arbeitet `length` auf allen zulässigen Typen gleich!

Beachte: Pascal oder C haben keinen parametrischen Polymorphismus
⇒ für jeden Listentyp muss eine eigene `length`-Funktion (mit identischer Struktur) definiert werden

Beispiel: Identität:

```
id x = x
```

Allgemeinster Typ: `id:: a → a`
Also: `id` auf jedes Argument anwendbar, aber es gilt immer: Ergebnistyp = Argumenttyp
⇒ `id` $\underbrace{\texttt{[1]}}_{\texttt{[Int]}}$ `==` $\underbrace{\texttt{['a']}}_{\texttt{[Char]}}$ Typfehler
$\underbrace{\phantom{\texttt{[Int]}}}_{\texttt{[Int]}}$

Unter **Typinferenz** verstehen wir die Herleitung allgemeinster Typen, so dass das Programm noch *typkorrekt* ist.

Vorteile:

1. Viele Programmierfehler werden als Typfehler entdeckt
   Beispiel: Vertauschung von Argumenten:

   ```
   foldr 0 (+) [1, 3, 5]   → Type error ...
   ```

2. Auch, falls kein Typfehler auftritt, kann der allgemeine Typ auf Fehler hinweisen:

   ```
   revf [] = []
   revf (x:xs) = revf xs ++ x   -- statt [x]
   ```

   Allgemeinster Typ:

   ```
   revf :: [[a]]  → [a]
   ```

   Allerdings ist der Argumenttyp `[[a]]` nicht beabsichtigt!

Was bedeutet aber **typkorrekt**?
Hier gibt es keine allgemeine Definition, sondern dies ist durch die jeweilige Programmiersprache festgelegt. Es gibt aber die allgemeine Forderung:

> „Typkorrekte“ Programme haben keine Laufzeittypfehler.

Im folgenden: **Typkorrektheit á la Hindley/Milner** [Milner 78, Damas/Milner 82] (beachte: Haskell basiert auf einer Verallgemeinerung mit „Typklassen“, deren genaue Definition aber wesentlich komplexer ist). Wir benötigen zur präzisen Definition zunächst einige grundlegende Begriffe.

Typausdrücke τ werden gebildet aus:

1. *Typvariablen* (a, b, c, ...)
2. *Basistypen* (Bool, Int, Float, Char, ...)
3. *Typkonstruktoren* (dies sind Operationen auf Typen, die aus gegebenen Typen neue bilden), wie z.B. $\cdot \to \cdot$, `[·]`, `Tree` $\cdot$

Beispiel: $\tau =$ `Int` $\to$ `[Tree Int]`

Wir unterscheiden:
monomorpher Typ: enthält keine Typvariablen
polymorpher Typ: enthält Typvariablen

(Typ-)**Substitution:** Ersetzung von Typvariablen durch Typausdrücke
Notation:

$$\sigma = \{a_1 \mapsto \tau_1, \ldots, a_n \mapsto \tau_n\}$$

bezeichnet eine Abbildung Typvariablen → Typausdrücke mit der Eigenschaft

$$\sigma(a) = \begin{cases} \tau_i & \text{falls } a = a_i \\ a & \text{sonst} \end{cases}$$

Fortsetzung dieser Abbildung auf Typausdrücken:

$\sigma(b) = b \quad \forall$ Basistypen b

$\sigma(k(\tau_1, \ldots, \tau_n)) = k(\sigma(\tau_n), \ldots, \sigma(\tau_n))$
$\forall$ n-stelligen Typkonstruktoren k und Typen $\tau_1, \ldots, \tau_n$

Beispiel: Falls $\sigma = \{\texttt{a} \mapsto \texttt{Int}, \texttt{b} \mapsto \texttt{Bool}\}$, dann ist $\sigma(\texttt{a} \to \texttt{b} \to \texttt{a}) = \texttt{Int} \to \texttt{Bool} \to \texttt{Int}$

Polymorphismus von Funktionen können wir nun wie folgt charakterisieren:

> Falls eine Funktion den (polymorphen) Typ $\tau_1 \to \tau_2$ hat und σ eine beliebige Substitution ist, dann hat sie auch den speziellen Typ $\sigma(\tau_1 \to \tau_2)$. Den spezielleren Typ bezeichnet man auch als **Typinstanz**.

Beispiel:

```
length :: [a]  → Int
```

hat auch die Typen

```
length :: [Int]   → Int    (σ = {a ↦ Int})
length :: [Char]  → Int    (σ = {a ↦ Char})
```

Es gilt in dem von uns betrachteten Typsystem die folgende Einschränkung:

> Keine Typinstanzbildung innerhalb der Definition derselben Funktion

Beispiel:
`f :: a -> a`
`f =` $\underbrace{\texttt{f}}_{(b\to b)\to(b\to b)}$ $\underbrace{\texttt{(f)}}_{b\to b}$ ist unzulässig!

Ursache für diese Einschränkung: die Typinferenz wäre sonst unentscheidbar

Daher: Unterscheide Funktion *mit* und *ohne* Instanzbildung. Dazu definieren wir:

Typschema: $\forall a_1 \ldots a_n : \tau$
(mit: $a_1, \ldots, a_n$: Typvariablen, τ: Typausdruck)
Beachte: für $n = 0$ ist jedes Typschema auch ein Typausdruck

generische Instanz des Typschemas $\forall a_1 \ldots a_n : \tau$: $\sigma(\tau)$ wobei $\sigma = \{a_1 \mapsto \tau_1, \ldots, a_n \mapsto \tau_n\}$

Vordefinierte Funktionen (d.h. alle Funktionen, deren Definition festliegt) haben Typschemata, während die zu definierenden Funktionen Typausdrücke haben.

Vorgehen bei Typprüfung

1. Rate für die zu definierenden Funktionen und deren Parameter Typausdrücke und prüfe (s.u.) alle Regeln für diese.
2. Bei Erfolg: Fasse diese Funktionen von nun an als vordefiniert auf, d.h. abstrahiere die Typvariablen zu einem Typschema.

Voraussetzung: Parameter in verschiedenen Regeln haben verschiedene Namen (dies können wir immer durch Umbenennung erreichen).

Typannahme A: Zuordnung Namen $\to$ Typschemata

Typprüfung: Beweisen von Aussagen der Form $A \vdash e :: \tau$ („Unter der Typannahme A hat der Ausdruck e den Typ τ“)

Beweis durch das folgende Inferenzsystem (Lese $\frac{P_1 \ldots P_n}{Q}$ so: falls $P_1 \ldots P_n$ richtig ist, dann auch Q)

Axiom $\quad \dfrac{}{A \vdash x :: \tau}$ falls τ generische Instanz von $A(x)$

Applikation $\quad \dfrac{A \vdash e_1 :: \tau_1 \to \tau_2, \quad A \vdash e_2 :: \tau_1}{A \vdash e_1\ e_2 :: \tau_2}$

Abstraktion $\quad \dfrac{A[x \mapsto \tau] \vdash e :: \tau'}{A \vdash \backslash x \to e :: \tau \to \tau'}$ wobei τ Typausdruck

$$A[x \mapsto \tau](y) = \begin{cases} \tau & \text{, falls } y = x \\ A(y) & \text{, sonst} \end{cases}$$

Bedingung $\quad \dfrac{A \vdash e_1 :: \texttt{Bool}, \quad A \vdash e_2 :: \tau, \quad A \vdash e_3 :: \tau}{A \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 :: \tau}$

Eine Gleichung $l = r$ ist **typkorrekt** bzgl. $A :\Leftrightarrow A \vdash l :: \tau$ und $A \vdash r :: \tau$ für einen Typausdruck τ

Falls alle Gleichungen für alle zu definierenden Funktionen aus A typkorrekt sind, abstrahiere deren Typen zu Typschemata (durch Quantifizierung der Typvariablen)
Beispiel: Es seien die folgenden vordefinierten Funktionen gegeben:

$$\begin{array}{lcl} A(\texttt{+}) & = & \texttt{Int} \to \texttt{Int} \to \texttt{Int} \\ A(\texttt{[]}) & = & \forall a.[a] \\ A(\texttt{:}) & = & \forall a.a \to [a] \to [a] \\ A(\texttt{not}) & = & \texttt{Bool} \to \texttt{Bool} \end{array}$$

Die Funktion `twice` sei wie folgt definiert:

```
twice f x = f (f x)
```

Wir erweitern unsere Typannahme wie folgt:

$$\begin{array}{lcl} A(f) & = & a \to a \\ A(x) & = & a \\ A(\texttt{twice}) & = & (a \to a) \to a \to a \end{array}$$

Damit können wir die Typkorrektheit wie folgt nachweisen:

Typ der linken Seite:

$$\dfrac{\dfrac{\overline{A \vdash \texttt{twice} :: (a \to a) \to a \to a} \quad \overline{A \vdash f :: a \to a}}{A \vdash \texttt{twice}\ f :: a \to a} \quad \overline{A \vdash x : a}}{A \vdash \texttt{twice}\ f\ x :: a}$$

Typ der rechten Seite:

$$\dfrac{\overline{A \vdash f : a \to a} \quad \dfrac{\overline{A \vdash f :: a \to a} \quad \overline{A \vdash x :: a}}{A \vdash (f\ x) :: a}}{A \vdash f\ (f\ x) :: a}$$

$\Longrightarrow$ Die Gleichung ist typkorrekt

Abstrahiere den geratenen Typ zu einem Typschema: `twice` $:: \forall a : (a \to a) \to a \to a$

Nun ist die polymorphe Anwendung von `twice` möglich:

```
twice (+2) 3
twice not True
```

Beachte: wenn in Haskell ein Funktionstyp definiert wird, dann wird dies implizit immer als Typschema interpretiert, bei dem die darin vorkommenden Typvariablen allquantifiziert sind. Somit entspricht das obige Typschema für `twice` der Typdeklaration

```
twice :: (a → a) → a → a
```

in Haskell.

τ heißt **allgemeinster Typ** eines Objektes (Funktion), falls

- τ ein korrekter Typ ist, und
- ist τ' ein korrekter Typ für dasselbe Objekt, dann gilt $\tau' = \sigma(\tau)$ für eine Typsubstitution σ.

Beispiel:

```
loop 0 = loop 0
```

Allgemeinster Typ:

```
loop:: Int → a
0 :: Int
```

Mit dieser Typananahme ist die Gleichung für `loop` typkorrekt:

$$\dfrac{\overline{A \vdash \ \texttt{loop} :: \texttt{Int} \to a} \quad \overline{A \vdash 0 :: \texttt{Int}}}{A \vdash \texttt{loop}\ 0 :: a}$$

⇒ Beide Gleichungsseiten haben Typ a

Der Typ `Int → a` deutet auf Programmierfehler hin, da die Funktion beliebige Ergebnisse erzeugen kann, was aber nicht möglich ist.

Die Typprüfung für mehrere Funktionen erfolgt analog, wobei wir nacheinander die Funktionen typisieren und dann deren Typausdrücke zu Typschemata abstrahieren. Beispiel:

```
length []     = 0
length (x:xs) = 1 + length xs

f xs ys = length xs + length ys
```

Vordefiniert seien die folgenden Symbole:

```
[]   :: ∀ a. [a]
(:)  :: ∀ a. a→ [a] → [a]
0, 1 :: Int
(+)  :: Int → Int → Int
```

Typannahme:

```
length :: [a] → Int
x      :: a
xs     :: [a]
```

Wir können zeigen, dass unter dieser Typannahme die Regeln für `length` wohlgetypt sind. Damit können wir nun `length` als vordefiniert mit folgendem Typschema annehmen:

```
length:: ∀a. [a]  →Int
```

Weitere Typannahme:

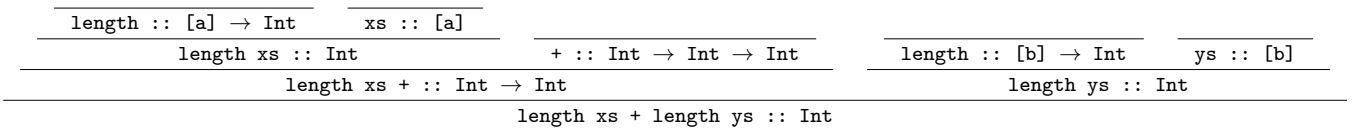
```
f  :: [a] → [b] → Int
xs :: [a]
ys :: [b]
```

In den folgenden Ableitungen lassen wir „$A \vdash$“ weg:

Mit dieser Typannahme erhalten wir folgende Ableitung für den Typ der linken Seite der Definition von `f`:

$$\dfrac{\dfrac{\dfrac{}{\texttt{f :: [a] → [b] → Int}} \quad \dfrac{}{\texttt{xs :: [a]}}}{\texttt{f xs :: [b] → Int}} \quad \dfrac{}{\texttt{ys :: [b]}}}{\texttt{f xs ys :: Int}}$$

In ähnlicher Weise erhalten wir eine Ableitung für den Typ der rechten Seite der Definition von `f` (hier sollte man beachten, dass hier zwei verschiedene generische Instanzen des Typschemas von `length` verwendet werden):



Somit erhalten wir das folgende Typschema für `f`:

```
f :: ∀a,b : [a] → [b] → Int
```

Daher ist

```
f [0,1] "Hallo"                                    (*)
```

wohlgetypt (mit der generischen Instanz $\{\texttt{a} \mapsto \texttt{Int}, \texttt{b} \mapsto \texttt{Char}\}$).

Ohne Typschemabildung für `length` würden wir für `f` den Typ `[a] → [a] → Int` erhalten, der zu speziell ist, d.h. mit dem der Ausdruck `(*)` nicht typkorrekt wäre.

Aus diesem Verfahren ergeben sich unmittelbar die folgenden Fragen zur praktischen Anwendung:

- Wie findet man allgemeinste Typen?
- Wie muss man Typannahmen raten?

Diese Fragen sollen im nächsten Kapitel beantwortet werden.

1.5.2 Typinferenz

Die Aufgabe der Typinferenz kann wie folgt formuliert werden:

> Suche (bzgl. vordefinierter Funktionen) die allgemeinsten Typen neu definierter Funktionen.

Nachfolgend zeigen wir die Grundidee der Typinferenz, wie sie z.B. in [Damas/Milner 82] zu finden ist.

Die Aufgabe der Typinferenz ist ja das Raten einer allgemeinsten Typannahme, so dass die definierten Funktionen unter der Typannahme typkorrekt sind. Statt nun den Typ zu raten, gehen wir wie folgt vor:

- Setze für einen unbekannten Typ zunächst eine neue Typvariable ein
- Formuliere Gleichungen (Bedingungen) zwischen Typen
- Berechne allgemeinste Lösung für das (Typ-)Gleichungssystem

Zunächst zeigen wir die Typinferenz für *eine* neue Funktion und erweitern dies später auf die Typinferenz für ganze Programme.

Vorgehen bei der Typinferenz:

1. **Variablenumbenennung:** Benenne Variablen in verschiedenen Gleichungen so um, dass verschiedene Gleichungen keine identischen Varablen enthalten:

   ```
   app [] x = x
   app (x:xs) y = x : app xs y
   ```

   wird umbenannt in

   ```
   app [] z = z
   app (x:xs) y = x : app xs y
   ```

2. **Erzeuge Ausdruck/Typ-Paare:**
   Für jede Regel

   l `|` c `=` r

   (wobei die Bedingung c auch fehlen kann) erzeuge die Ausdruck/Typ-Paare

   $l :: a$
   $r :: a$
   $c :: \texttt{Bool}$

   wobei a eine *neue* Typvariable ist

3. **Vereinfache Ausdruck/Typ-Paare:** Ersetze
   - $(e_1\ e_2) :: \tau$ durch $e_1 :: a \to \tau,\ e_2 :: a$ (hierbei ist a eine neue Typvariable)
   - `if` e_1 `then` e_2 `else` $e_3 :: \tau$ durch $e_1 :: \texttt{Bool}, e_2 :: \tau, e_3 :: \tau$
   - $e_1 \circ e_2 :: \tau$ durch $e_1 :: a, e_2 :: b, o :: a \to b \to \tau$ (a, b neue Typvariablen)
     (hierbei ist $\circ$ ein vordefinierter Infixoperator)
   - $\backslash x \to e :: \tau$ durch $x :: a, e :: b, \underbrace{\tau \doteq a \to b}_{Typgleichung}$ (a, b neue Typvariablen)
   - $f :: \tau$ durch $\tau \doteq \sigma(\tau')$
     falls f vordefiniert mit Typschema $\forall a_1 \ldots a_n : \tau'$ und
     $\sigma = \{a_1 \mapsto b_1, \ldots, a_n \mapsto b_n\}$
     wobei $b_1, \ldots b_n$ neue Typvariablen sind.

   Die hierbei erzeugten Typgleichungen werden zusammen mit den im nächsten Schritt erzeugten Typgleichungen aufgesammelt.

4. **Erzeuge (Typ-)Gleichungen:**
   Nach dem vorigen Schritt können wir nun voraussetzen, dass alle Ausdruck/Typ-Paare von der Form $x :: \tau$ (mit x Bezeichner) sind.

   Dann: setze alle Typen für einen Bezeichner gleich, d.h. erzeuge die Gleichung $\tau_1 \doteq \tau_2$ für alle Ausdruck/Typ-Paare $x :: \tau_1$, $x :: \tau_2$

Beispiel: betrachte die Definition von `twice`:

```
twice f x = f (f x)
```

Anwendung von Schritt 2 ergibt:

`twice f x` $:: a$
`f (f x)` $:: a$

Anwendung von Schritt 3:

`twice f x` $:: a \quad \Rightarrow \quad$ `twice f` $:: b \to a$, `x` $:: b$
`twice f` $:: b \to a \quad \Rightarrow \quad$ `twice` $:: c \to b \to a$, `f` $:: c$

`f (f x)` $:: a \quad \Rightarrow \quad$ `f` $:: d \to a$, `(f x)` $:: d$
`(f x)` $:: d \quad \Rightarrow \quad$ `f` $:: e \to d$, `x` $:: e$

Insgesamt haben wir am Ende also die Ausdruck/Typ-Paare:

`x` $:: b$
`twice` $:: c \to b \to a$
`f` $:: c$
`f` $:: d \to a$
`f` $:: e \to d$
`x` $:: e$

Daraus erzeugen wir die folgenden Typgleichungen:

$c \doteq d \to a$
$c \doteq e \to d$
$b \doteq e$

5. Löse nun das sich ergebende Gleichungssystem, d.h. finde einen „allgemeinsten Unifikator“ σ hierfür (s.u.). Dann ist, für alle Ausdruck/Typ-Paare $x :: \tau$, $\sigma(\tau)$ der allgemeinste Typ für x. Falls kein allgemeinster Unifikator existiert, dann sind die Regeln nicht typkorrekt.

   Somit gilt:

   > Für typkorrekte Regeln existiert immer ein allgemeinster Typ!

   Diese Aussage ist nicht trivial, z.B. ist sie falsch, wenn man ad-hoc-Polymorphismus mit Overloading betrachtet.

Zur Lösung des letzten Punktes, d.h. die Berechnung eines allgemeinsten Unifikators, definieren wir zunächst: Definition:

- Eine Substitution σ heißt **Unifikator** für ein Gleichungssystem E, falls für alle Gleichungen $l \doteq r \in E$ gilt: $\sigma(l) = \sigma(r)$
- Ein Unifikator σ für ein Gleichungssystem E heißt **allgemeinster Unifikator** (**mgu**, **most general unifier**) für E, falls für alle Unifikatoren σ' für E eine Substitution φ existiert mit $\sigma' = \varphi \circ \sigma$ (wobei $\varphi \circ \sigma(\tau) = \varphi(\sigma(\tau))$), d.h. alle anderen Unifikatoren sind Spezialfälle von σ.

Satz 1.2 ([Robinson 65]) *Falls ein Unifikator existiert, dann existiert auch ein mgu, der effektiv berechenbar ist.*

Hier: mgu-Berechnung nach [Martelli/Montanari 82] durch Transformation von E:

Wende folgende Transformationsregeln auf E an:

Decomposition:	$\dfrac{\{k\ s_1 \dots s_n \doteq k\ t_1 \dots t_n\} \cup E}{\{s_1 \doteq t_1, \dots, s_n \doteq t_n\} \cup E}$	k Typkonstruktor
Clash:	$\dfrac{\{k\ s_1 \dots s_n \doteq k'\ t_1 \dots t_m\} \cup E}{\texttt{fail}}$	$k \neq k'$ Typkonstruktoren
Elimination:	$\dfrac{\{x \doteq x\} \cup E}{E}$	x Typvariable
Swap:	$\dfrac{\{k\ t_1 \dots t_n \doteq x\} \cup E}{\{x \doteq k\ t_1 \dots t_n\} \cup E}$	x Typvariable
Replace:	$\dfrac{\{x \doteq \tau\} \cup E}{\{x \doteq \tau\} \cup \sigma(E)}$	x Typvariable, kommt in E aber nicht in τ vor, $\sigma = \{x \mapsto \tau\}$
Occur check:	$\dfrac{\{x \doteq \tau\} \cup E}{\texttt{fail}}$	x Typvariable, $x \neq \tau$, x kommt in τ vor

Satz 1.3 *Falls E mit den obigen Transformationen in $\{x_1 \doteq \tau_1, \dots, x_n \doteq \tau_n\}$ überführbar und dann keine weitere Regel anwendbar ist, dann ist $\sigma = \{x_1 \mapsto \tau_1, \dots, x_n \mapsto \tau_n\}$ ein mgu für E. Falls E in `fail` überführbar ist, dann existiert kein Unifikator für E.*

Beispiel: Wir betrachten die Berechnung eines mgus für unser obiges Gleichungssystem:

$$\dfrac{\dfrac{\dfrac{\dfrac{\boxed{c \doteq d \to a},\ c \doteq e \to d,\ b \doteq e}{c \doteq d \to a,\ \boxed{d \to a \doteq e \to d},\ b \doteq e}\ \text{Replace}}{c \doteq d \to a,\ \boxed{d \doteq e},\ a \doteq d,\ b \doteq e}\ \text{Decomposition}}{c \doteq e \to a,\ d \doteq e,\ \boxed{a \doteq e},\ b \doteq e}\ \text{Replace}}{c \doteq e \to e,\ d \doteq e,\ a \doteq e,\ b \doteq e}\ \text{Replace}$$

Daher ist

$$\sigma = \{c \mapsto e \to e,\ d \mapsto e,\ a \mapsto e,\ b \mapsto e\}$$

ein allgemeinster Unifikator für das Gleichungssystem und

$$\texttt{twice} :: (e \to e) \to e \to e$$

ein allgemeinster Typ. Somit ist `twice` typkorrekt und hat das Typschema

$$\forall e.(e \to e) \to e \to e$$

Typinferenz für beliebige rekursive Funktionen

Idee:

1. Sortiere Funktionen nach Abhängigkeiten
2. Inferiere zunächst den Typ der Basisfunktionen. Falls diese typkorrekt sind, wandle deren Typen zu Typschemata um.
3. Inferiere die darauf aufbauenden Funktionen. Falls diese typkorrekt sind, wandle deren Typen zu Typschemata um.
4. ...

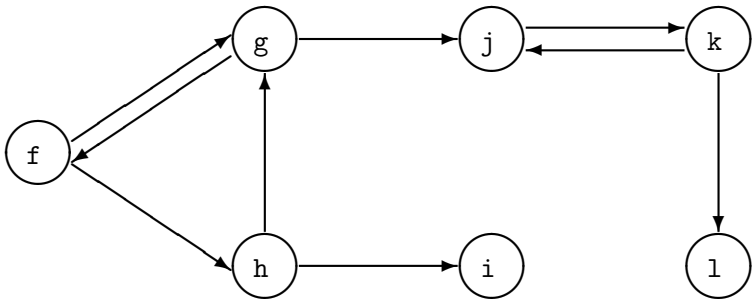
Präsisierung durch einen **statischen Aufrufgraph**:

- Knoten: Funktionen
- Kante $f \to g$ existiert genau dann, wenn g in der Definition von f verwendet wird.

Beispiel:

```
f n  =  g n + h n
g n  =  j n + f n
h n  =  i n + g n
i n  =  n
j n  =  k n
k n  =  j n + l n
l n  =  5
```

Statischer Aufrufgraph für dieses Beispiel:



Die Relation „gegenseitig abhängig“ wird dann wie folgt definiert:
$f \leftrightarrow g :\Leftrightarrow$ es existiert ein Pfad von f nach g und ein Pfad von g nach f

Feststellung: $\leftrightarrow$ ist eine Äquivalenzrelation

Daher können wir die Äquivalenzklassen bzgl. $\leftrightarrow$ bilden (diese werden auch **starke Zusammenhangskomponenten** genannt). Im obigen Beispiel sind dies z.B. die Knotenmengen $\{\texttt{f}, \texttt{g}, \texttt{h}\}$ und $\{\texttt{j}, \texttt{k}\}$.

Sortiere nun die Äquivalenzklassen in eine Liste

$$[A_1, A_2, \ldots, A_n]$$

wobei gilt: wenn ein Pfad von A_i-Knoten nach A_j-Knoten ($i \neq j$) existiert (d.h. A_i ist mit Hilfe von A_j definiert), dann ist $i \geq j$.

Sortierung für das obigen Beispiel:

$$[\{\texttt{i}\}, \{\texttt{l}\}, \{\texttt{j}, \texttt{k}\}, \{\texttt{f}, \texttt{g}, \texttt{h}\}]$$

Nun tue das Folgende für alle Listenelemente A_i ($i = 1, \ldots, n$):

1. Berechne den allgemeinsten Typ für alle Funktionen in A_i.
2. Abstrahiere diese Typen durch Allquantifizierung aller vorkommenden Typvariablen.

Beispiel:

```
i     :: ∀a : a → a
l     :: ∀a : a → Int
j,k   :: ∀a : a → Int
f,g,h :: Int → Int
```

Ein Beispiel für eine nicht typisierbare Funktion ist

```
f x = x x
```

(der Ausdruck "`x x`" wird auch als **Selbstanwendung** bezeichnet). Für dieses Beispiel ergibt das Typinferenzverfahren nach der Vereinfachung der Ausdruck/Typ-Paare:

$\texttt{f}::b \to a,\ \texttt{x}::b,\ \texttt{x}::c \to a,\ \texttt{x}::c$

Nunn stellen wir das Gleichungssystem auf und berechnen den mgu:

$$\dfrac{\dfrac{b \doteq c \to a,\ \boxed{b \doteq c}}{\boxed{c \doteq c \to a},\ b \doteq c}\ \text{Replace}}{\texttt{fail}}\ \text{Occur check}$$

Somit ist dieses Gleichungsssytem nicht lösbar und damit die Funktion `f` nicht typisierbar.

Nicht typisierbar sind aber auch sinnvolle Funktionsanwendungen. Betrachten wir dazu die Funktion

```
funsum f xs ys = f xs + f ys
```

und den Ausdruck

```
funsum length [1,2,3] "abc"
```

In einer ungetypten Sprache würde hierfür das Ergebnis `6` berechnet werden. Bei dem hier dargestellten Typsystem führt dieser Ausdruck allerdings zu einem **Typfehler**!

Die Ursache liegt im inferierten Typ von `funsum`:

$\forall a : (a \to \texttt{Int}) \to a \to a \to \texttt{Int}$

Bei Aufruf von `funsum` muss man eine generische Instanz festlegen, z.B. $a =$ `[Int]`, was aber zu einem Typfehler bei `"abc"` führt.

Ein geeigneter Typ wäre:

$\texttt{funsum}\ \forall b, c : (\forall a : a \to \texttt{Int}) \to b \to c \to \texttt{Int}$

d.h. der erste Parameter ist eine polymorph verwendbare Funktion.
Solche Typen sind prinzipiell denkbar ($\rightsquigarrow$ Polymorphismus 2. Ordnung), aber in dem hier vorgestellten Typsystem gilt:

Parameter sind nicht mehrfach typinstanziierbar

Praktisch ist dies aber kein Problem, da mehrfache Aufrufe wegtransformiert werden können:

```
funsum :: (a  → Int)  → (b  → Int)  → a  → b  → Int
funsum  f1 f2 l1 l2 = f1 l1 + f2 l2
```

Mit dieser Definition ist der Ausdruck

```
funsum length length [1,2,3] "abc"
```

typisierbar und ergibt `6`.

1.6 Lazy Evaluation

In diesem Kapitel wollen wir die besondere Auswertungstrategie moderner funktionaler Sprachen genauer betrachten. Dazu benötigen wir zunächst einige Begriffe:

Redex (*re*ducible *ex*pression): Ein Ausdruck (Funktionsaufruf), auf den die linke Seite einer Gleichung passt (vgl. Kapitel 1.3).

Normalform: Ein Ausdruck ohne Redex (dies entspricht einem „Wert“).

Reduktionsschritt: Ersetze einen Redex durch eine entsprechend passende rechte Gleichungsseite

Das *Ziel einer Auswertung*:

> Berechne Normalform durch Anwendung von Reduktionsschritten

Das Problem hierbei ist: Ein Ausdruck kann viele Redexe enthalten! Welche Redexe soll man reduzieren?

Alternativen:

- „Sichere“ Strategie: Reduziere alle Redexe $\rightsquigarrow$ zu aufwändig
- Strikte Sprachen: Reduziere linken inneren Redex (*LI-Reduktion*)
- Nichtstrikte Sprachen: Reduziere linken äußeren Redex (*LO-Reduktion*)
  $\rightarrow$ Nachteil: Mehrfachauswertung von Argumenten

Beispiel:

```
double x = x+x

> double (1+2)  →  (1+2) + (1+2)  →  3 + (1+2)  →  3+3  →  6
```

Vermeidung durch *Graphrepräsentation:* Mehrfache Vorkommen von Variablen duplizieren keine Terme, sondern werden als Verweise auf einen Ausdruck interpretiert:

```
double (1+2) →  · + ·   →  · + ·  →  6
                ↓ ↓        ↓ ↓
                (1+2)       3
```

Die LO-Reduktion auf Graphen wird auch als **lazy evaluation** oder **call-by-need**-Reduktion bezeichnet.

Charakteristik:

1. Berechne einen Redex nur, falls er „benötigt“ wird
2. Berechne jeden Redex höchstens einmal

Wann wird nun ein Redex „benötigt“?

1. Durch ein Muster:

   ```
   f (x:xs) = ...
   ```

   Wenn wir nun den Aufruf

   ```
   f (g...)
   ```

   betrachten, dann wird der Wert (Konstruktor) von `(g...)` benötigt. Genauer wird dies ausgedrückt durch die Übersetzung von Mustern in case-Ausdrücke (vgl. Kapitel 1.3). Präziser:

   > Bei der Auswertung von “`case` t `of...`” wird der Wert von t benötigt.

2. Durch strikte Grundoperationen: Arithmetische Operationen und Vergleiche benötigen den Wert ihrer Argumente
   Dagegen: `if` e_1 `then` $e2$ `else` e_3 benötigt nur den Wert von e_1
3. Ausgabe: Drucken der benötigten Teile

Beim Fall 1 ist allerdings folgendes zu beachten:

> Der benötigte Redex wird nicht komplett ausgewertet, sondern nur bis zur **schwachen Kopfnormalform** (**SKNF**) (**weak head normal form**, **WHNF**).

Eine schwache Kopfnormalform ist hierbei eines der folgenden Möglichkeiten:

- $b \in$ `Int` $\cup$ `Float` $\cup$ `Bool` $\cup$ `Char` (primitiver Wert)
- $C\ e_1 \ldots e_k$, wobei C ein n-stelliger Konstruktor ist und $k \leq n$
- $(e_1, \ldots, e_k)$ (Tupel)
- $\backslash x \rightarrow e$ (lambda-Abstraktion)
- $f\ e_1 \ldots e_k$, wobei f eine n-stellige Funktion ist und $k < n$ (partielle Applikation)

Ein SKNF-Ausdruck ist also kein Redex, d.h. nicht weiter (an der Wurzel) reduzierbar.

Wichtig: Beim Pattern matching ($\simeq$ case-Ausdrücke) muss nur reduziert werden, bis eine schwache Kopfnormalform erreicht ist!
Bsp: Betrachte die Operation `take` n xs: berechne die ersten n Elemente von xs

```
take 0     xs     = []
take (n+1) (x:xs) = x : take n xs
```

```
take 1 ([1,2] ++[3,4])
          benötigt
→ take 1 (1:([2] ++ [3,4]))
              Redex
→ 1 : take 0 ([2] ++ [3,4])
            Redex
→ 1 : []   (≃ [1])
```

Beachte: das 2. Argument wurde nicht komplett ausgewertet (daher der Name „lazy“ evaluation).

Lazy evaluation hat Vorteile bei sog. *nicht-strikten* Funktionen (Vermeidung von überflüssigen Berechnungen):
Eine n-stellige Funktion f heißt **strikt** im i-ten Argument ($1 \leq i \leq n$), falls für alle $x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n$ gilt:

$$f\ x_1 \ldots x_{i-1}\ \bot\ x_{i-1} \ldots x_n = \bot$$

Hierbei bezeichnet $\bot$ eine nichtterminierende Berechnung (undefinierter Wert)
Z.B. ist

```
if-then-else :: Bool → a → a → a
```

strikt im 1. Argument, aber nicht strikt im 2. und 3. Argument

Vorteile von lazy evaluation:

- Vermeidung überflüssiger Auswertungen
  ($\leadsto$ *optimale Reduktionen*, [Huet/Lévy 79, Huet/Lévy 91])
- Rechnen mit unendlichen Datenstrukturen
  ($\leadsto$ Trennung von Daten und Kontrolle, $\leadsto$ Modularisierung, vgl. [Hughes 90])

Unendliche Datenstrukturen

Betrachten wir die Erzeugung einer (endlichen) Liste von ganzen Zahlen:

```
fromTo f t | f > t     = []
           | otherwise = f : fromTo (f + 1) t

fromTo 1 5  ⇝ [1, 2, 3, 4, 5]
```

Falls „`t` $= \infty$“ ist, dann ist das Ergebnis eine unendliche Liste, weil die Bedingung `f > t` nie wahr wird.

Spezialisierung von `fromTo` für `t` $= \infty$:

```
from f = f : from (f + 1)
```

```
from 1  ⇝ 1 : 2 : 3 : 4 : ... ^C
```

Obwohl `from 1` ein unendliches Objekt darstellt, kann man es wie andere Listen verwenden:

```
take 5 (from 1)  ⇝ [1, 2, 3, 4, 5]
```

Somit ist `fromTo` ein Spezialfall von `from`:

```
fromTo f t = take (t-f+1) (from f)
```

Vorteil: klare Trennung von Kontrolle (`take (t-f+1)`) und Datenerzeugung (`(from f)`)

Die Kontrolle und Datenerzeugung ist dagegen in der ersten Version von `fromTo` vermischt.

Häufig ist die unabhängige Datenerzeugung auch einfacher definierbar, wie wir unten an Beispielen zeigen werden.

Beachte: die Auswertung unendlicher Datenstrukturen wird erst durch lazy evaluation möglich! Betrachten wir dazu den Ausdruck

```
take 2 (from 1)
```

Bei einer strikten Auswertung (eager evaluation) wird versucht, das Argument `(from 1)` vollständig auszuwerten, was zu einer Endlosschleife führt. Dagegen geht eine lazy Berechnung wie folgt vor:

`take 2` $\underbrace{\texttt{(from 1)}}_{benötigt}$
→ `take 2` $\underbrace{\texttt{(1 : from (1+1))}}_{in\ SKNF}$
→ `1 : take 1` $\underbrace{\texttt{(from (1+1))}}_{benötigt}$
→ `1 : take 1 ((1+1) : from ((1+1)+1))`
→ `1 : (1+1) : take 0 (from ((1+1)+1))`
→ `1 : (1+1) : []`
→ `1 : 2 : []`

Beispiel: Erzeugung aller **Fibonacci-Zahlen**
Hierbei ist die n. Fibonacci-Zahl die Summe der beiden vorherigen
⇒ Im Gegensatz zu `from` benötigen wir hier zwei Parameter (bei beiden vorherigen Fibonacci-Zahlen)

```
fibgen n1 n2 = n1 : fibgen n2 (n1 + n2)
```

```
fibs = fibgen 0 1
```

$\Rightarrow$ lineare (statt exponentielle) Laufzeit!

```
fibs  ⇝  0 : 1 : 1: 2 : 3 : 5 : 8 ...
```

Nun können wir die gewünschte Kontrolle hinzufügen, wie z.B.:
Berechne alle Fibonacci-Zahlen < 50:

```
takeWhile (<50) fibs  ⇝ [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Achtung: Filtern hier nicht möglich:

```
filter (<10) fibs   ⇝ [0, 1, 2, 3, 5, 8,... Endlosschleife!
```

Beispiel: Primzahlberechnung durch *Sieb des Eratosthemes*:
Idee:

1. Erstelle die Liste aller natürlicher Zahlen ≥ 2
2. Streiche darin alle Vielfachen bereits gefundener Primzahlen
3. Nächste kleinste Zahl ist prim

Beispiel:
$\underline{2}, \underline{3}, \not{4}, \underline{5}, \not{6}, \underline{7}, \not{8}, \not{9}, \not{10}, \underline{11}, \not{12}, \underline{13}, \ldots$
Wir realisieren dies mit Hilfe einer Funktion

```
sieve :: [Int]  → [Int]
```

Hierbei gelten die folgende Invarianten für `sieve`:

1. Das erste Element x der Eingabeliste ist prim und die restliche Eingabeliste enthält keine Vielfachen von Primzahlen $< x$.
2. Die Ergebnisliste besteht nur aus Primzahlen.

Mit diesen Überlegungen kann `sieve` wie folgt implementiert werden:

```
sieve :: [Int]  → [Int]
sieve (x:xs) = x : sieve (filter (\y  → y `mod` x > 0) xs)
```

Hierbei liefert das Prädikat in `filter` den Wert `False`, falls `y` ein Vielfaches von `x` ist. Insgesamt erhalten wir unter Benutzung von `sieve` die folgende Definition für die Liste aller Primzahlen:

```
primes :: [Int]
primes = sieve (from 2)
```

Die ersten 100 Primzahlen können wir dann wie folgt berechnen:

```
take 100 primes
```

Beispiel: **Hamming-Zahlen**: Dies ist eine aufsteigende Folge von Zahlen der Form $2^a * 3^b * 5^c, a, b, c, \geq 0$
d.h. alle Zahlen mit Primfaktoren $2, 3, 5$

Idee:

1. 1 ist kleinste Hamming-Zahl
2. Falls n Hamming-Zahl, dann ist auch $2n$, $3n$, $5n$ eine Hamming-Zahl
3. Wähle aus den nächsten möglichen Hamming-Zahlen die kleinste aus und gehe nach 2.

Realisierung: Erzeuge die unendliche Liste der Hamming-Zahlen durch Multiplikation der Zahlen mit $2, 3, 5$ und mische die multiplizierten Listen aufsteigend geordnet.

Mischen zweier unendlicher(!) Listen in aufsteigender Folge:

```
ordMerge (x:xs) (y:ys)  |  x==y   =  x : ordMerge xs     ys
                        |  x < y  =  x : ordMerge xs     (y:ys)
                        |  x > y  =  y : ordMerge (x:xs) ys
```

Mischen dreier Listen `l1`, `l2`, `l3` durch "`ordMerge l1 (ordMerge l2 l3)`"
Somit:

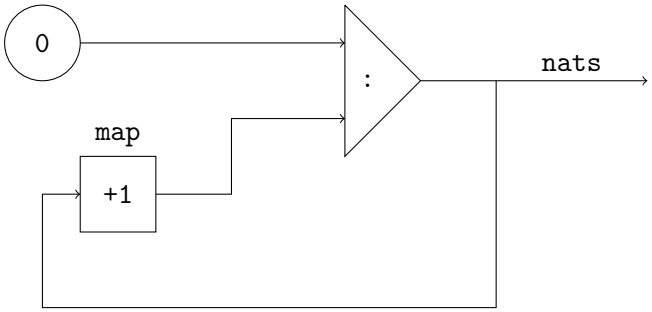
```
hamming = 1 : ordMerge (map (2*) hamming)
                       (ordMerge (map (*3) hamming)
                                 (map (*5) hamming))
```

Effizienz: `ordMerge` und `map` haben einen konstanten Aufwand für die Berechnung des nächsten Elements, so dass der Aufwand $0(n)$ für die Berechnung des n. Elements ist.

Berechne die ersten 15 Hammingzahlen durch:

```
take 15 hamming  ⇝ [1,2,3,4,5,6,8,9,10,12,15,16,18,20,24]
```

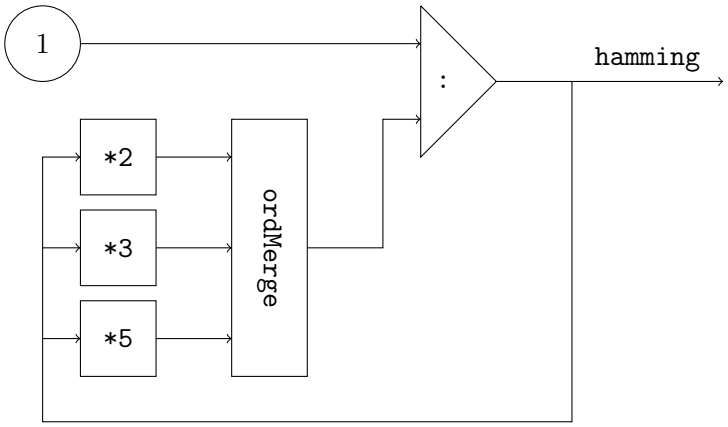
Da unendliche Listen viel Ähnlichkeit mit Strömen haben, kann man das Rechnen mit unendlichen Listen gut mittels *Prozessnetzen* planen. Z.B. entspricht dem Strom natürlicher Zahlen das folgende Netzwerk:



Hieraus ergibt sich die Haskell-Definition

```
nats = 0 : map (+1) nats
```

In ähnlicher Weise können wir das Netzwerk für Hamming-Zahlen beschreiben:



1.7 Deklarative Ein/Ausgabe

Wie kann man die Ein-/Ausgabe von Daten in einer deklarativen Sprache realisieren?

Eine Möglichkeit: Seiteneffekte wie in imperativen Sprachen (dies wird z.B. in Standard ML oder Scheme gemacht), z.B.

```
getInt :: Int
```

Seiteneffekt bei Auswertung von `getInt`: Lesen einer Zahl von der Eingabe

Problematisch: Abhängigkeit von der Auswertungsreihenfolge, z.B.

```
inputDiff = getInt - getInt
```

Hier ist in der Regel `inputDiff` verschieden von 0!

Außerdem ist das Ergebnis abhängig von der Auswertungsreihenfolge von "`-`", z.B.
Eingabe: `5 3`
Ergebnis: entweder `2` oder `-2`!

Bei Lazy Evaluation ist die genaue Auswertungsreihenfolge schwer zu überschauen, daher ist hier ein I/O-Konzept ohne Seiteneffekte besonders wichtig!

Idee einer deklarativen Ein/Ausgabe: fasse I/O-Operationen als „Aktionen“ auf, die die „äußere Welt“ (vom Typ `World`) verändern/transformieren. Dies könnten wir durch folgenden Typ ausdrücken:

```
type IO a = World  → (a,World)
```

Somit verändert eine *Aktion* vom Typ `IO a` den Weltzustand und liefert dabei ein Element vom Typ `a`.

Beispiele:

```
getChar :: IO Char        -- liest ein Zeichen
putChar :: Char  → IO () -- schreibt ein Zeichen
```

Der Ergebnistyp `()` deutet an, dass `putChar` kein Ergebnis liefert, sondern nur den Weltzustand verändert.

Wichtig: Der Datentyp `World` bezeichnet den Zustand der gesamten äußeren Welt, er ist aber für den Programmierer nicht zugreifbar (insbesondere kann dieser nicht kopiert und auf zwei verschiedene Arten weiter transformiert werden!). Der Programmierer hat lediglich die Möglichkeit, Basisaktionen (wie z.B. `getChar`) zu größeren Einheiten zusammenzusetzen.

Komposition von Aktionen:
Da Aktionen Seiteneffekte auf der Welt sind, ist deren Ausführungsreihenfolge sehr wichtig. Aus diesem Grund ist es nur erlaubt, Aktionen sequenziell zusammenzusetzen. Hierzu gibt es einen Operator “`>>=`”:

```
(>>=) :: IO a  → (a  → IO b)  → IO b
```

„Definition“ dieser Operation:

```
(a >>= fa) world = case a world of
                     (x,world')  → fa x world'
```

Somit passiert durch die Abarbeitung von `(a >>= fa)` folgendes:

1. Führe Aktion `a` auf der aktuellen Welt aus.
2. Dies liefert ein Ergebnis `x` sowie eine veränderte Welt.
3. Führe die Aktion `(fa x)` auf der veränderten Welt aus.
4. Dies liefert ein Ergebnis `x'`, was auch das Gesamtergebnis ist.

Beispiel:

```
getChar >>= putChar
```

kopiert ein Zeichen von der Eingabe auf die Ausgabe.

Spezialfall: Komposition ohne Ergebnisverarbeitung:

```
(>>) :: IO a  → IO b  → IO b
a1 >> a2 = a1 >>= (\_  → a2)
```

„Leere Aktion“:

```
return :: a  → IO a -- mache nichts, liefere nur das Ergebnis
```

Beispiel: Ausgabe eines Strings (diese Aktion ist in Haskell so vordefiniert):

```
putStr :: String  → IO ()
putStr []     = return ()
putStr (c:cs) = putChar c >> putStr cs
```

Damit lautet das typische “Hello World”-Programm in Haskell:

```
main = putStr "Hello, World!"
```

Damit man I/O-Aktionen schöner aufschreiben kann, gibt es als „syntaktischen Zucker“ die `do`-**Notation**:

```
do p <- a1
   a2
```

steht für “`a1 >>= \p -> a2`”, und

```
do a1
   a2
```

steht für “`a1 >> a2`” (wobei nach dem “`do`” die Layout-Regel zur Anwendung kommt!).

Beispiel: Kopieren einer Eingabezeile:

```
echoLine = do c <- getChar
              putChar c
              if c=='\n' then return ()
                         else echoLine
```

Und nun noch einmal die Definition von `inputDiff`:

```
inputDiff = do i1 <- getInt
               i2 <- getInt
               putStr (show (i1-i2))
```

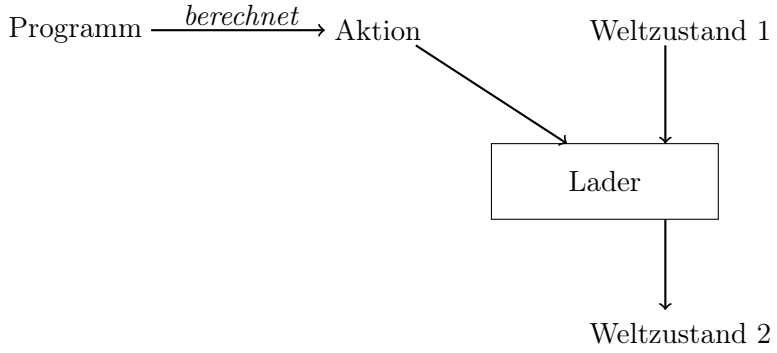
Hier ist die Reihenfolge der Berechnung explizit gemacht und damit nicht abhängig von der Auswertungsreihenfolge von "`-`".

Somit ist die Grundidee der deklarativen Ein/Ausgabe:

> Ein interaktives Programm berechnet eine Aktion(sfolge), diese verändert die Welt, *wenn diese auf einen Weltzustand angewendet wird.*

Aber wann passiert das?

Bei Ausführung des Programms durch das Betriebssystem! Daher wendet der Programmlader (Betriebssystem) die Aktion auf die momentane Welt an:



Vorteile dieses Konzepts:

- keine Seiteneffekte
- Aktionsreihenfolge ist explizit
- funktionale (referenziell transparente) Berechnung komplexer Aktionen möglich

Beispiel:

```
dupAct a = a >> a
```

Ausführung von `dupAct (putStr "Ha")` ergibt die Ausgabe "`HaHa`".
Dagegen in Standard ML:

```
fun dupAct a = (a ; a)
```

Ausdruck `dupAct (print "Ha")` ergibt die Ausgabe "`Ha`".

Somit können durch die deklarative Ein/Ausgabe die üblichen funktionalen Programmiertechniken auch zum Rechnen mit Aktionen benutzt werden.

1.8 Zusammenfassung

Vorteile funktionaler Sprachen:

- einfaches Modell: Ersetzung von Ausdrücken
- überschaubare Programmstruktur:
  Funktionen, definiert durch Gleichungen für Spezialfälle (pattern matching)
- Ausdrucksmächtigkeit: Programmschemata mittels Funktionen höherer Ordnung ausdrückbar
- Sicherheit: keine Laufzeittypfehler
- Wiederverwendung: Polymorphismus + Funktion höherer Ordnung
- Modularität und Optimalität durch Lazy Evaluation
- Verifikation und Wartbarkeit:
  keine Seiteneffekte: erleichtert Verifikation und Transformation

2 Grundlagen der funktionalen Programmierung

In diesem Kapitel wollen wir wichtige Begriffe der deklarativen Programmierung formalisieren. Hierbei sind die folgenden Bereiche relevant:

Reduktionssysteme: hiermit werden ganz allgemein Begriff wie Reduktionsfolgen, Normalformen, Terminierung u.ä. definiert.

Termersetzung: hiermit werden das Rechnen mit Pattern matching und passende Auswertungsstrategien definiert.

2.1 Reduktionssysteme

Rechnen in funktionalen Sprachen kann aufgefasst werden als das fortlaufende Anwenden von Reduktionsschritten. Um hierfür einige allgemeine Begriffe festzulegen, betrachten wir hier *allgemeine (abstrakte) Reduktionssysteme*. Die damit einhergehenden Begriffe werden z.B. beim λ-Kalkül, Termersetzung und der Logikprogrammierung wiederverwendet.

Definition 2.1 (Reduktionssystem) *Sei M eine Menge und $\to$ eine binäre (zweistellige) Relation auf M, d.h. $\to \subseteq M \times M$ und wir schreiben $e_1 \to e_2$ falls $(e_1, e_2) \subseteq \to$. In diesem Fall heißt $(M, \to)$* ***Reduktionssystem****.*
Wir definieren verschiedene Erweiterungen der Relation $\to$:

- $x \to^0 y :\Leftrightarrow x = y$
- $x \to^i y :\Leftrightarrow \exists z : \quad x \to^{i-1} z$ *und* $z \to y$ *(i-faches Produkt von $\to$)*
- $x \to^+ y :\Leftrightarrow \exists i > 0 : \; x \to^i y$ *(transitiver Abschluss)*
- $x \to^* y :\Leftrightarrow \exists i \geq 0 : \; x \to^i y$ *(reflexiv-transitiver Abschluss)*

Außerdem definieren wir von $\to$ abgeleitete Relationen:

- $x \leftarrow y :\Leftrightarrow \quad y \to x$
- $x \leftrightarrow y :\Leftrightarrow \quad x \to y$ *oder* $x \leftarrow y$

Insbesondere ist $x \leftrightarrow^* y$ der symmetrische, reflexiv-transitive Abschluss bzw. die kleinste Äquivalenzrelation, die $\to$ enthält.

Definition 2.2 *Sei $(M, \to)$ ein festes Reduktionssystem.*

- $x \in M$ ***reduzierbar*** $:\Leftrightarrow \exists y$ *mit* $x \to y$
- $x \in M$ ***irreduzibel*** *oder* ***Normalform****, falls x nicht reduzierbar ist.*
- y ***Normalform von*** $x \;:\Leftrightarrow\; x \to^* y$ *und y Normalform (irreduzibel). Falls x nur eine Normalform y hat, schreiben wir auch:* $x \downarrow := y$
- $x \downarrow y$ *(x und y sind* ***zusammenführbar****)* $:\Leftrightarrow \exists z$ *mit* $x \to^* z$ *und* $y \to^* z$

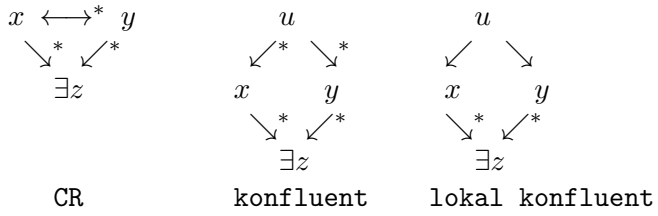
Wichtig: Funktionen sollten „sinnvoll“ definiert sein, d.h. zu berechnende Ausdrücke sollten höchstens eine Normalform haben, damit das Ergebnis eindeutig ist.

Hierzu muss das Reduktionssystem weitere Forderungen erfüllen:

Definition 2.3 *Sei $(M, \to)$ ein Reduktionssystem.*

1. $\to$ *heißt* ***Church-Rosser*** *(CR), falls* $\leftrightarrow^* \subseteq \downarrow$ *(d.h.* $x \leftrightarrow^* y$ *impliziert* $x \downarrow y$*).*
2. $\to$ *heißt* ***konfluent****, falls* $\forall u, x, y$ *mit* $u \to^* x$ *und* $u \to^* y$ *ein z existiert mit* $x \to^* z$ *und* $y \to^* z$*.*
3. $\to$ *heißt* ***lokal konfluent****, falls* $\forall u, x, y$ *mit* $u \to x$ *und* $u \to y$ *ein z existiert mit* $x \to^* z$ *und* $y \to^* z$

Graphisch:



Intuition:

Church-Rosser: Die Äquivalenz von Ausdrücken kann auch durch gerichtete Reduktionen entschieden werden.

Konfluenz: Unterschiedliche Berechnungswege spielen keine Rolle.

Lokale Konfluenz: Unterschiedliche lokale Berechnungswege spielen keine Rolle.

Es ist aus der Definition der Konfluenz unmittelbar klar, dass eine konfluente Reduktionsrelation garantiert, dass jedes Element höchstens eine Normalform hat. Aus diesem Grund ist die *Konfluenz wünschenswert* für die effiziente Implementierung funktionaler Sprachen (da man dann nicht in beliebiger Richtung nach Normalformen suchen muss). Allerdings ist „lokal konfluent“ i.allg. leichter prüfbar als „konfluent“. Daher stellt sich die Frage, welche Zusammenhänge es zwischen diesen Begriffen gibt. Der folgende Satz gibt darüber Auskunft:

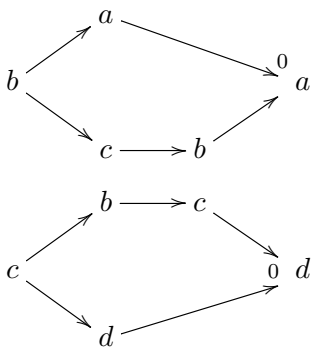
Satz 2.1

1. *„$\to$ ist CR“ genau dann wenn „$\to$ ist konfluent“*
2. *Aus „$\to$ ist konfluent“ folgt „$\to$ ist lokal konfluent“*

Die Umkehrung von 2 gilt leider nicht:

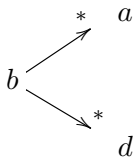
Beispiel:
Sei $\to$ definiert durch $b \to a$
$b \to c$
$c \to b$
$c \to d$
Graphisch: $a \leftarrow b \leftrightarrows c \to d$

Bei diesem Reduktionssystem gibt es zwei lokale Divergenzen:



Da alle lokalen Divergenzen zusammenführbar sind, ist das Reduktionssystem lokal konfluent.

Das Reduktionssystem ist aber nicht konfluent:



aber es gilt nicht $a \downarrow d$.
Das Problem in diesem Beispiel ist die „Schleife“ zwischen b und c. Daher definieren wir zunächst Reduktionssysteme ohne solche Schleifen:

Definition 2.4 *Sei $(M, \to)$ ein Reduktionssystem.*

1. *$\to$ heißt **terminierend** oder **Noethersch**, falls keine unendlichen Ketten*

   $$x_1 \to x_2 \to x_3 \to \dots$$

   existieren.

2. $\rightarrow$ *heißt* ***konvergent*** *(eindeutig terminierend), falls* $\rightarrow$ *konfluent und terminierend ist.*

Für terminierende Reduktionssysteme gilt der folgende wichtige Satz:

Satz 2.2 (Newman-Lemma) *Sei* $(M, \rightarrow)$ *terminierend. Dann gilt:*

$\rightarrow$ *konfluent* $\Leftrightarrow$ $\rightarrow$ *lokal konfluent*

Somit gilt: Für terminierende Reduktionssysteme ist Konfluenz „einfach“ prüfbar:

> Betrachte alle 1-Schritt-Divergenzen (evtl. sind dies nur endlich viele) und berechne die Normalformen der Elemente (diese existieren wegen Terminierung). Falls diese jeweils identisch sind, ist das Reduktionssystem (lokal) konfluent.

Aber leider sind i.allg. Reduktionen in funktionalen Sprachen nicht immer terminierend (z.B. bei Verwendung unendlicher Datenstrukturen). Aus diesem Grund werden wir später andere Kriterien zur Konfluenz kennenlernen.

2.2 Termersetzungssysteme

Eine Motivation für die Entwicklung der Theorie der Termersetzungssysteme war die Präzisierung des Rechnens mit Gleichungsspezifikationen.

Beispiel: Betrachten wir die aus der Mathematik bekannten Gruppenaxiome:

$$\begin{array}{rcll} x * 1 & = & x & (1) \\ x * x^{-1} & = & 1 & (2) \\ (x * y) * z & = & x * (y * z) & (3) \end{array}$$

Rechnen: Ersetze Gleiches durch Gleiches (auch in beliebiger Richtung!)

Als Beispiel wollen wir zeigen, dass für alle Gruppen gilt: $1 * x = x$

$$\begin{array}{rcll} 1 * x & = & (x * x^{-1}) * x & (2) \\ & = & x * (x^{-1} * x) & (3) \\ & = & x * (x^{-1} * (x * 1)) & (1) \\ & = & x * (x^{-1} * (x * (x^{-1} * (x^{-1})^{-1}))) & (2) \quad \text{für } x^{-1} \text{ statt } x \\ & = & x * (x^{-1} * ((x * x^{-1}) * (x^{-1})^{-1})) & (3) \\ & = & x * (x^{-1} * (1 * (x^{-1})^{-1})) & (2) \\ & = & x * ((x^{-1} * 1) * (x^{-1})^{-1}) & (3) \\ & = & x * (x^{-1} * (x^{-1})^{-1}) & (1) \\ & = & x * 1 & (2) \\ & = & x & (1) \end{array}$$

Aspekte:

- Termstrukturen, Anwendungen von Gleichungen in Teiltermen
- „Pattern matching“: Ersetzen von Variablen in Gleichungen durch andere Terme
- Gleichungen in beide Richtungen anwenden

Termersetzung als Modell zum Rechnen mit Funktionen:

- Gleichungen nur von links nach rechts anwenden
- Einschränkung der linken Seiten, damit effektive Strategien möglich sind

Nachfolgend werden wir daher die notwendigen Details genau definieren.

Definition 2.5 *Wir legen die folgenden Grundbegriffe für Termersetzungssysteme fest:*

- ***Signatur**: $\Sigma = (S, F)$ mit*
  - *S: Menge von **Sorten** ($\approx$ Basistypen Nat, Bool,...)*
  - *F: Menge von **Funktionssymbolen** $f :: s_1, \ldots, s_n \to s, n \geq 0$*
    *($c ::\to s$ heißen auch **Konstanten**)*
- ***Variablen** haben auch Sorten, d.h. eine Variablenmenge*

  $$V = \{x :: s \mid x \text{ Variablensymbol}, s \in S\}$$

  hat die Eigenschaft $x :: s, x :: s' \in V \Rightarrow s = s'$ (kein Überladen bei Variablen erlaubt)
- *Die Menge der **Terme** über Σ und V der Sorte s wird mit $T(\Sigma, V)_s$ bezeichnet und ist wie folgt definiert:*
  - *$x \in T(\Sigma, V)_s$ falls $x :: s \in V$*
  - *$f(t_1, \ldots, t_n) \in T(\Sigma, V)_s$ falls $f :: s_1, \ldots, s_n \to s \in F$ (wobei $\Sigma = (S, F)$) und $t_i \in T(\Sigma, V)_{s_i}$ $(i = 1, \ldots, n)$*
    Notation: Schreibe c statt $c()$ für Konstanten
- *Die Menge aller Variablen in einem Term t wird mit $\mathcal{V}ar(t)$ bezeichnet und ist definiert durch*

  $$\begin{aligned} \mathcal{V}ar(x) &= \{x\} \\ \mathcal{V}ar(f(t_1, \ldots, t_n)) &= \mathcal{V}ar(t_1) \cup \ldots \cup \mathcal{V}ar(t_n) \end{aligned}$$

- *t heißt **Grundterm** falls $\mathcal{V}ar(t) = \varnothing$*
- *t heißt **linear** falls keine Variable in t mehrfach vorkommt.*
  Bsp.: $f(x, y)$ ist linear, $f(x, x)$ ist nicht linear.

- *Ein **Termersetzungssystem** (**TES**) ist eine Menge von **Regeln** der Form $l \to r$ mit $l, r \in T(\Sigma, V)_s$ (für ein $s \in S$) und $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$.*
  *Hierbei wird auch l als **linke Seite** und r als **rechte Seite** der Regel bezeichnet.*

Beispiel:
Addition auf natürliche Zahlen (die wiederum aus 0 und s(uccessor) aufgebaut sind):

$$S = \texttt{Nat}, F = \{0 ::\to \texttt{Nat}, s :: \texttt{Nat} \to \texttt{Nat}, + :: \texttt{Nat,Nat} \to \texttt{Nat}\}$$

$$\begin{array}{lrcl} \text{Regeln:} & 0 + n & \to & n \\ & s(m) + n & \to & s(m + n) \end{array}$$

Intuitive Semantik dieses Termersetzungssystems: Regeln sind Gleichungen, die jedoch nur von links nach rechts angewendet werden.

Ziel: Beschreibung des Rechnens mit Termersetzungssystemen, wie z.B. Reduktion

$$s(0) + s(0) \to s(s(0))$$

Dazu notwendig: Anwendung einer Regel (*Substitution*) an einer beliebigen *Position*:

Definition 2.6 (Substitution, Position, Teilterm, Ersetzung)

- ***Substitution** $\sigma : V \to T(\Sigma, V)$ mit $\sigma(x) \in T(\Sigma, V)_s$ für alle $x :: s \in V$, wobei der Definitionsbereich*

  $$\mathcal{D}om(\sigma) = \{x \mid \sigma(x) \neq x\}$$

  endlich ist. Notation:

  $$\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$$

  bezeichnet die Substitution

  $$\sigma(x) = \begin{cases} t_i & \text{falls } x = x_i \\ x & \text{sonst} \end{cases}$$

  Wir können eine Substitution einfach auf Terme erweitern oder fortsetzen:
  $\sigma : T(\Sigma, V) \to T(\Sigma, V)$ wird definiert durch

  $$\sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n))$$

- ***Positionen** in einem Term sind Bezeichner für Teilterme, ausgedrückt durch Zahlenfolgen.*
  $\mathcal{P}os(t)$ bezeichnet die Menge aller Positionen im Term t:
  - *$\epsilon \in \mathcal{P}os(t)$ (Wurzelposition, wobei ϵ: leere Folge)*
  - *Falls $p \in \mathcal{P}os(t_i)$, dann ist auch $i \cdot p \in \mathcal{P}os(f(t_1, \ldots, t_n))$*

- *Der **Teilterm** bzw. **Unterterm** von t an der Position $p \in \mathcal{P}os(t)$ wird mit $\underline{t|_p}$ bezeichnet und ist wie folgt definiert:*

$$\begin{aligned} t|_\epsilon &= t \\ f(t_1, \ldots, t_n)|_{i \cdot p} &= t_i|_p \end{aligned}$$

- *Die **Ersetzung** des Teilterms an der Position p im Term t durch s wird mit $\underline{t[s]_p}$ bezeichnet und ist wie folgt definiert:*

$$\begin{aligned} t[s]_\epsilon &= s \\ f(t_1, \ldots, t_n)[s]_{i \cdot p} &= f(t_1, \ldots, t_{i-1}, t_i[s]_p, t_{i+1}, \ldots, t_n) \end{aligned}$$

- *Zum Vergleich von Positionen verwenden wir die folgenden Begriffe:*
  - *$p \leq q$ (p ist **über** q) falls p Präfix von q, d.h.] $\exists r$ mit $p \cdot r = q$*
  - *p und q sind **disjunkt** $:\Leftrightarrow$ es gilt weder $p \leq q$ noch $q \leq p$*
  - *p heißt **links** von $q :\Leftrightarrow p = p_0 \cdot i \cdot p_1$ und $q = p_0 \cdot j \cdot q_1$ mit $i < j$*

Nun können wir die Ersetzungsrelation bzgl. eines TES R definieren:

Definition 2.7 (Termersetzung) *Sei R ein TES. Dann ist die Reduktionsrelation (bzgl. R)*

$$\rightarrow_R \ \subseteq \ T(\Sigma, V) \times T(\Sigma, V)$$

wie folgt definiert:

$t_1 \rightarrow_R t_2 :\Leftrightarrow \exists l \rightarrow r \in R$, *Position* $p \in \mathcal{P}os(t_1)$ *und Substitution* σ *mit* $t_1|_p = \sigma(l)$ *und* $t_2 = t_1[\sigma(r)]_p$

*$t_1 \rightarrow_R t_2$ wird auch **Reduktionsschritt** oder **Ersetzungsschritt** genannt. $\sigma(l)$ ist eine „passende“ linke Regelseite. In diesem Fall wird $t_1|_p$ auch **Redex** genannt.*

Notation:

$t_1 \rightarrow_{p, l \rightarrow r} t_2$	falls Position und Regel wichtig
$t_1 \rightarrow t_2$	falls R fest gewählt ist

Beispiel:

$$\begin{aligned} R: \quad 0 + n &\rightarrow n && (1) \\ s(m) + n &\rightarrow s(m + n) && (2) \end{aligned}$$

Für dieses Termersetzungssystem gibt es folgende Reduktionsschritte:

$$\begin{aligned} s(s(0)) + s(0) &\rightarrow_{\varepsilon,(2)} s(s(0) + s(0)) \\ &\rightarrow_{1,(2)} s(s(0 + s(0))) \\ &\rightarrow_{1 \cdot 1,(1)} s(s(s(0))) \end{aligned}$$

Der letzte Term $s(s(s(0)))$ ist in Normalform, d.h. nicht weiter reduzierbar.

Für sinnvoll definierte funktionale Programme ist es wünschenswert, dass Normalformen *eindeutig* sind. In Kapitel 2.1 haben wir gesehen, dass die Konfluenz bzw. Church-Rosser-Eigenschaft hinreichend für eindeutige Normalformen ist.
Tatsächlich ist das obige +-Beispiel konfluent, wohingegen das Gruppenbeispiel nicht konfluent ist. Um letzteres zu sehen, betrachte die beiden Regeln:

$$\begin{array}{rcll} x * x^{-1} & \rightarrow & 1 & (1) \\ (x * y) * z & \rightarrow & x * (y * z) & (2) \end{array}$$

Dann gibt es für den Term $(a * a^{-1}) * b$ die folgenden Reduktionsschritte:

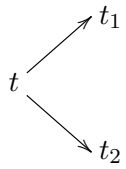
$$\begin{array}{ll} (a * a^{-1}) * b & \rightarrow_{\epsilon,(2)} \quad a * (a^{-1} * b) \\ (a * a^{-1}) * b & \rightarrow_{1,(1)} \quad 1 * b \end{array}$$

Die jeweils letzten Terme sind verschiedene Normalformen des Terms $(a * a^{-1}) * b$. Daher sind die Regeln nicht konfluent.

Diese Beispiele zeigen, dass Konfluenz stark abhängig vom jeweiligen Termersetzungssystem ist. Gibt es also hinreichende Kriterien, die z.B. durch einen Compiler überprüft werden könnten?

Hierzu haben Knuth und Bendix gezeigt ([Knuth/Bendix 70]), dass es ausreicht, sogenannte **kritische Paare** (d.h. Überlappungen linker Seiten) zu betrachten.

Idee: Falls es eine lokale Divergenz

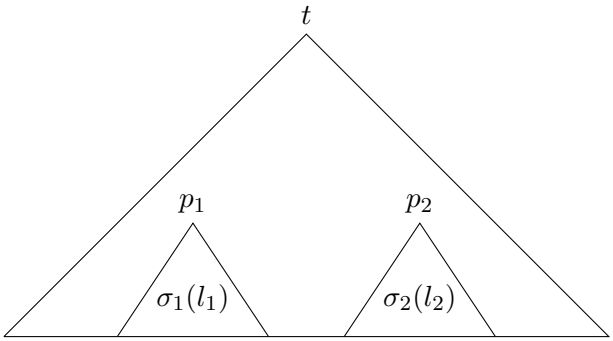


gibt, dann muss auf Grund der Definition von Termersetzungsschritten gelten:

$\exists p_i, \sigma_i, l_i \rightarrow r_i \in R$ mit $t|_{p_i} = \sigma_i(l_i)$ und $t_i = t[\sigma_i(r_i)]p_i($ für $i = 1, 2)$

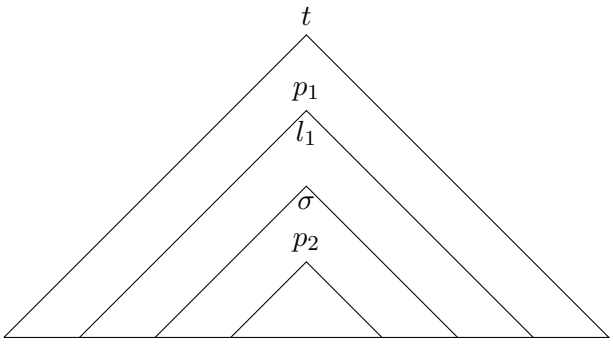
Unterscheide dann die folgenden Fälle:

- p_1 und p_2 sind disjunkt:



Führe t_1 und t_2 zusammen durch Anwendung von $l_i \to r_i$ an Stelle p_i in t_{3-i} $(i = 1, 2)$

- p_1 über p_2, aber nicht innerhalb von l_1:



  Dann sind t_1 und t_2 ebenfalls zusammenführbar (beachte: p_2 liegt innerhalb einer Variablen in l_1; somit $l_2 \to r_2$ auch anwendbar in allen diesen Variablen in r_2)

- p_2 über p_1 , aber nicht innerhalb von l_2: analog

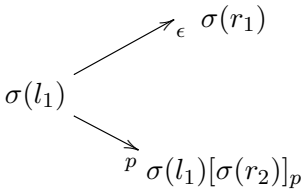
- Falls keiner der vorigen Fälle zutrifft, haben wir eine „echte“ Überlappung von l_1 und l_2, die wir nicht nicht einfach zusammenführen können. Daher spricht man in diesem Fall von einem *kritischen Paar*.

Definition 2.8 *Seien $l_1 \to r_1, l_2 \to r_2 \in R$ mit $Var(l_1) \cap Var(l_2) = \varnothing$, $p \in Pos(l_1)$ mit $l_1|_p \notin V$ und σ mgu (vgl. Kapitel 1.5.2, Typinferenz) für $l_1|_p$ und l_2. Falls $p = \epsilon$, dann sei auch $l_1 \to r_1$ keine Variante (gleich bis auf Variablenumbenennung) von $l_2 \to r_2$. Dann heißt*

$$\langle \sigma(r_1),\ \sigma(l_1)[\sigma(r_2)]_p \rangle$$

kritisches Paar *von $l_1 \to r_1$ und $l_2 \to r_2$.*
Weiterhin bezeichnen wir mit $CP(R)$ die Menge aller kritischer Paare aus R.

Erklärung:



ist eine kritische Divergenz (da $\sigma(l_1)|_p = \sigma(l_2)$)

Satz 2.3 (Kritisches-Paar-Lemma [Knuth/Bendix 70]) *Sei R ein TES.*
Falls $t \to t_1$ und $t \to t_2$, dann ist $t_1 \downarrow_R t_2$ oder $t_1 \leftrightarrow_{CP(R)} t_2$.

D.h. alle echten lokalen Divergenzen kommen von kritischen Paaren. Somit haben wir eine Charakterisierung der lokalen Konfluenz mittels Zusammenführung kritischer Paare:

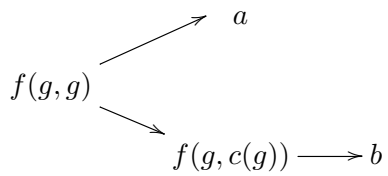
Satz 2.4 *Sei R TES. Dann gilt: R lokal konfluent $\Leftrightarrow t_1 \downarrow_R t_2 \; \forall \langle t_1, t_2 \rangle \in CP(R)$*

Falls also R endlich ist und $\rightarrow_R$ terminierend, so können wir die Konfluenz durch Zusammenführung aller kritischen Paare entscheiden.

Kann man dieses Kriterium auch bei nichtterminierenden TES verwenden?
Leider nicht, denn dann ist die Untersuchung kritischer Paare nicht ausreichend. Betrachte hierzu das folgende Termersetzungssystem R:

$$\begin{array}{rcl} f(x,x) & \rightarrow & a \\ f(x,c(x)) & \rightarrow & b \\ g & \rightarrow & c(g) \end{array}$$

Offensichtlich gilt hier $CP(R) = \varnothing$, d.h. es existieren keine kritischen Paare. Somit ist R lokal konfluent, aber nicht konfluent:



Beachte: Die linke Seite der ersten Regel ist nicht-linear!

Definition 2.9 *Ein TES R mit l linear $\forall l \rightarrow r \in R$ („links-linear“) heißt*

- ***orthogonal*** *falls $CP(R) = \varnothing$*
- ***schwach orthogonal*** *falls für alle $\langle t, t' \rangle \in CP(R)$ gilt: $t = t'$*
  (d.h. es existieren nur triviale kritische Paare)

Es gilt:

Satz 2.5 *(Schwach) orthogonale TES sind konfluent.*

Somit verfügen wir nun über folgende hinreichende Kriterien für Konfluenz:

- bei terminierenden TES: Zusammenführung kritischer Paare
  (die Eigenschaft „terminierend“ ist teilweise prüfbar mit sog. *Terminierungsordnungen*)
- im Allg.: Prüfe Linkslinearität und Trivialität von Überlappungen

Problem: Wie findet man die Normalform (falls sie existiert) eines Terms bzgl. eines konfluenten TES?

- Bei terminierenden TES: beliebige Folge von Reduktionsschritten
- Im Allgemeinen: durch eine geeignete Reduktionsstrategie

Im Folgenden sei nun immer R ein konfluentes Termersetzungssystem, falls dies nicht explizit erwähnt ist.

Definition 2.10 *Eine* ***Reduktionsstrategie*** S *ist ein Abbildung* $S : T(\Sigma, V) \to 2^{Pos(T(\Sigma,V))}$, *wobei* $S(t) \subseteq Pos(t)$ *mit* $t|_p$ *ist Redex* $\forall p \in S(t)$ *und* $\forall p_1, p_2 \in S(t)$ *mit* $p_1 \neq p_2$ *gilt:* p_1 *und* p_2 *sind disjunkt.*

Intuition: S legt die Positionen fest, bei denen als nächstes reduziert wird

Weiter sei:

- $t_1 \to_R t_2 \to_R t_3 \to_R \ldots$ *heißt* ***S-Reduktion****, falls für* $i = 1, 2, 3 \ldots$ *gilt:* t_{i+1} *entsteht aus* t_i *durch Anwendung von Reduktionsschritten an allen Positionen aus* $S(t_i)$.
- S *heißt* ***sequenziell*** $:\Leftrightarrow \forall t \in T(\Sigma, V)$ *gilt:* $|S(t)| \leq 1$
- S *heißt* ***normalisierend*** $:\Leftrightarrow \forall t, t' \in T(\Sigma, V)$ *mit* $t' = t \downarrow_R$ *gilt: jede S-Reduktion* $t \to_R t_1 \to_R \ldots$ *endet in* t' *(d.h. die Strategie S berechnet immer die Normalform, falls sie existiert).*

Beispiele für Reduktionsstrategien (im Folgenden nehmen wir an, dass $t \in T(\Sigma, V)$ nicht in Normalform ist):

Leftmost innermost (LI): $S(t) = \{p\}$ falls $t|_p$ Redex und $\forall q \neq p$ mit $t|_q$ Redex gilt: q rechts oder über p

Leftmost outermost (LO): $S(t) = \{p\}$ falls $t|_p$ Redex und $\forall q \neq p$ mit $t|_q$ Redex gilt: p über oder links von q

Parallel outermost (PO): $S(t) = \{p \mid t|_p$ Redex und $\forall q \neq p$ mit $t|_q$ Redex gilt: $q \not\leq p\}$

Die Strategie LI ist nicht normalisierend, wie man am folgenden Beispiel sieht:

$$\begin{array}{rrcl} R: & 0 * x & \to & 0 \\ & a & \to & a \end{array}$$

Es gilt $(0 * a) \downarrow_R = 0$, aber:

$$0 * \underline{a} \to_R^{LI} 0 * \underline{a} \to_R^{LI} 0 * a \ldots$$

Auch die Strategie LO ist nicht normalisierend für orthogonale TES:[1]

$$\begin{array}{rrcl} R: & a & \to & b \\ & c & \to & c \\ & f(x, b) & \to & d \end{array}$$

[1] Beachte: Die Strategie LO ist nicht identisch zur Pattern-Matching-Strategie aus Kapitel 1.3!

Auf Grund der Ableitung

$$f(c, \underline{a}) \;\to_R\; \underline{f}(c, b) \;\to_R\; d$$

sehen wir, dass $f(c, a) \downarrow_R = d$ gilt, aber es gibt nur eine LO-Ableitung:

$$f(\underline{c}, a) \;\to_R^{LO}\; f(\underline{c}, a) \;\to_R^{LO}\; f(\underline{c}, a) \ldots$$

Dagegen berechnet die PO-Strategie die Normalform:

$$f(\underline{c}, \underline{a}) \;\to_R^{PO}\; \underline{f}(c, b) \;\to_R^{PO}\; d$$

Allgemein gilt:

Satz 2.6 (O'Donnell [O'Donnell 77]) *PO ist normalisierend für (schwach) orthogonale Systeme.*

Aber: PO ist aufwändiger zu implementieren als LO.

Im vorigen Beispiel könnte man auch "Rightmost outermost" benutzen, um die Normalform zu berechnen. Im Allgemeinen ist eine sequenzielle Strategie nicht ausreichend, sondern manchmal ist die parallele Reduktion essentiell:

Beispiel: „Paralleles Oder":

```
R: true  ∨ x     → true
   x     ∨ true  → true
   false ∨ false → false
```

Gegeben: $t_1 \vee t_2$:
Gefahr bei einer sequenziellen Strategie: Falls versucht wird t_1 (oder t_2) zu reduzieren, könnte dies zu einer unendlichen Ableitung führen, wohingegen t_2 (oder t_1) zu `true` reduzierbar ist.

Somit wäre in diesem Beispiel ein „sicheres" Vorgehen, t_1 und t_2 parallel auszuwerten.

In diesem Beispiel ist R nicht orthogonal, da die ersten beiden Regeln überlappen. Daher stellt sich die Frage: Sind orthogonale Systeme sequenziell normalisierbar?

Leider gilt dies auch nicht immer, wie folgendes Beispiel von Berry zeigt:

$$\begin{array}{lll} R: & f(x,0,1) & \to \;\; 0 \\ & f(1,x,0) & \to \;\; 1 \\ & f(0,1,x) & \to \;\; 2 \end{array}$$

Dieses System ist orthogonal, aber wenn wir den Ausdruck $f(t_1, t_2, t_3)$ betrachten, ist es nicht klar, Welches t_i zuerst reduziert werden muss.

Aus diesen Betrachtungen stellt sich die interessante Frage:

> Gibt es für bestimmte Klassen von TES sequenzielle (und evtl. optimale) Reduktionsstrategien?

Definition 2.11 *Ein TES R heißt* ***linksnormal*** *(**left-normal**), falls für alle Regeln $l \to r \in R$ gilt: in l steht „hinter" einer Variablen kein Funktionssymbol.*

Beispiele:
Das System

$$\begin{array}{rcl} 0 * x & \to & 0 \\ a & \to & a \end{array}$$

ist linksnormal. Dagegen ist die Regel

$$f(x, b) \to d$$

nicht linksnormal.

Satz 2.7 (O'Donnell [O'Donnell 77]) *LO ist normalisierend für linksnormale orthogonale TES.*

Anwendung: Die Kombinatorlogik (eine Implementierung des Lambda-Kalküls) oder auch ein Termersetzungssysteme R mit $Var(l) = \varnothing$ für alle $l \to r \in R$ sind linksnormal, sodass hierfür LO normalisierend ist.

Als nächstes betrachten wir eine weitere wichtige Klasse von Termersetzungssystemen, für die auch gute Auswertungsstrategien existieren:

Definition 2.12 *Sei R ein TES bzgl. einer Signatur $\Sigma = (S, F)$.*

- *$f : s_1, \ldots, s_n \to s \in F$ heißt* ***definierte Funktion****, falls eine Regel*

  $$f(t_1, \ldots, t_n) \to r$$

  existiert. Die Menge aller definierten Funktionen ist dann wie folgt definiert:

  $$D = \{f | f \text{ definierte Funktion}\} \subseteq F$$

- *$C = F \backslash D$ heißt Menge der* ***Konstruktoren****.*
- *$f(t_1, \ldots, t_n)(n \geq 0)$ heißt* ***Muster*** *(**pattern**), falls $f \in D$ und $t_1, \ldots, t_n$ sind* ***Konstruktorterme****, d.h. sie enthalten keine definierte Funktionen.*
- *R heißt* ***konstruktorbasiert****, falls l ein Muster ist $\forall l \to r \in R$.*

Beispiel:

$$\begin{array}{rrcl} R: & 0 + n & \to & n \\ & s(m) + n & \to & s(m + n) \end{array}$$

Hier ist $D = \{+\}$, $C = \{0, s\}$ und R ist konstruktorbasiert.
Die obigen Gruppenaxiome sind dagegen nicht konstruktorbasiert.

Intuition:

- Konstruktoren bauen Datenstrukturen auf.
- Definierte Funktionen rechnen auf Datenstrukturen.
- konstrukturbasierte TES entsprechen funktionalen Programmen, jedoch ist hier keine Reihenfolge der Regeln festgelegt (was eher eine gleichungsorientierte Interpretation ist).

Induktiv-sequentielle Termersetzungssysteme [Antoy 92] haben die Eigenschaft, dass Funktionen induktiv über den Datenstrukturen definiert sind. Dies ist zum Beispiel bei der Additionsoperation + der Fall, die durch eine Fallunterscheidung (Induktion) über das erste Argument definiert ist.

Die genaue formale Definition ist wie folgt.

Definition 2.13 (Definierender Baum) *Ein **definierender Baum (definitional tree)** ist ein Baum, bei dem jeder Knoten mit einem Muster markiert ist. Dabei gibt es zwei Arten von definierenden Bäumen mit einem Muster π:*

- Regelknoten *der Form $l \to r$ mit $\pi = l$*
- Verzweigungsknoten *der Form $branch(\pi, p, \mathcal{T}_1, \ldots, \mathcal{T}_k)$, wobei gilt:*
  - *$p \in Pos(\pi)$ mit $\pi|_p \in V$*
  - *$\mathcal{T}_i$ $(i = 1, \ldots, k)$ ist ein definierender Baum mit dem Muster $\pi[C_i(x_1, \ldots, x_{m_i})]_p$, wobei $x_1, \ldots, x_{m_i}$ neue Variablen und $C_1, \ldots, C_k$ sind verschiedene Konstruktoren sind.*

Wir bezeichnen mit $pattern(\mathcal{T})$ das Muster des definierenden Baumes $\mathcal{T}$.

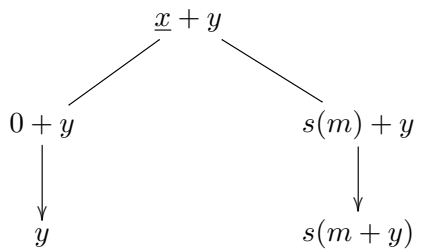
Definition 2.14 (Induktiv-sequenziell) *Sei R ein Termersetzungssystem.*

- *$\mathcal{T}$ heißt definierender Baum für die Funktion f, falls $\mathcal{T}$ endlich ist und das Muster $f(x_1, \ldots, x_n)$ hat ($x_1, \ldots, x_n$ sind verschiedene Variablen), jeder Regelknoten ist eine Variante einer Regel aus R, und jede Regel $f(t_1, \ldots, t_n) \to r \in R$ kommt in $\mathcal{T}$ genau einmal vor. In diesem Fall heißt f **induktiv-sequenziell**.*
- *R heißt **induktiv-sequenziell**, falls alle $f \in D$ induktiv-sequenziell sind.*

Beispiel: Betrachten wir noch einmal die Additionsfunktion:

$$\begin{array}{lrcl} R: & 0 + n & \to & n \\ & s(m) + n & \to & s(m + n) \end{array}$$

Definierender Baum (in graphischer Darstellung):



Dagegen ist das parallele Oder

```
true   ∨  x      →  true
x      ∨  true   →  true
false  ∨  false  →  false
```

nicht induktiv-sequentiell, da hier kein eindeutiges Argument für eine Fallunterscheidung existiert.

Satz 2.8 *Jedes induktiv-sequentielle TES ist orthogonal und konstruktorbasiert (aber nicht umgekehrt!)*

Für induktiv-sequentielle TES existiert eine einfache sequentielle Reduktionsstrategie, die Konstruktornormalformen berechnet:

Definition 2.15 *Die Reduktionsstrategie φ sei wie folgt definiert:*
Sei t ein Term, o die Position des linkesten äußersten definierten Funktionssymbols in t (d.h. $t|_o = f(t_1, \ldots, t_n)$ mit $f \in D$) und $\mathcal{T}$ ein definierender Baum für f. Dann ist:

$$\varphi(t) = \{o \cdot \varphi(t|_o, \mathcal{T})\}$$

Hier ist zu beachten, dass $\varphi(t|_o, \mathcal{T})$ eventuell undefiniert sein kann; in diesem Fall ist auch $\varphi(t)$ undefiniert. Weiterhin ist

$$\varphi(t, l \to r) = \epsilon \quad \textit{(d.h. wende Regel an)}$$

$$\varphi(t, branch(\pi, p, \mathcal{T}_1, \ldots, \mathcal{T}_k)) =$$

$$\begin{cases} \varphi(t, \mathcal{T}_j) & \textit{falls } t|_p = C_j(s_1, \ldots, s_{m_j}) \textit{ und } pattern(\mathcal{T}_j)|_p = C_j(x_1, \ldots, x_{m_j}) \\ p \cdot \varphi(t|_p, \mathcal{T}) & \textit{falls } t|_p = f(\ldots) \textit{ mit } f \in D \textit{ und } \mathcal{T} \textit{ def. Baum für } f \end{cases}$$

Intuitiv: Die Auswertung einer Funktion erfolgt durch Analyse des zugehörigen definierenden Baumes:

- Bei Regelknoten: Anwenden der Regel
- Bei Verzweigungsknoten: Aktueller Wert an Verzweigungsposition ist
  - Konstruktor $\rightsquigarrow$ betrachte entsprechenden Teilbaum
  - Funktion $\rightsquigarrow$ Werte Funktion aus

Beispiel:
Betrachte die Regeln für + (s.o.), den definierenden Baum $\mathcal{T}_+$ für + (wie oben dargestellt und den Term $t = (s(0) + 0) + 0$.
Dann wird der erste Auswertungsschritt für t wie folgt berechnet:

$$\begin{aligned} \varphi(t) &= \epsilon \cdot \varphi(t, \mathcal{T}_+) \\ &= 1 \cdot \varphi(s(0) + 0, \mathcal{T}_+) \\ &= 1 \cdot \varphi(s(0) + 0, s(m) + y \to s(m + y)) \\ &= 1 \cdot \epsilon = 1 \end{aligned}$$

Insgesamt ergibt sich folgende Auswertung bzgl. der Strategie φ:

$$\begin{array}{lll} t & \to_R^\varphi & s(0+0)+0 \\ & \to_R^\varphi & s((0+0)+0) \quad \text{da } \varphi(s(0+0)+0) = \epsilon \\ & \to_R^\varphi & s(0+0) \quad\quad\quad \text{da } \varphi(s((0+0)+0) = 1 \cdot 1 \\ & \to_R^\varphi & s(0) \quad\quad\quad\quad \text{da } \varphi(s(0+0)) = 1 \end{array}$$

Die Strategie φ berechnet evtl. nicht die Normalform eines Terms. Dies kann zwei Ursachen haben:

1. Term enthält Variablen:

   $$(x+0)+(0+0)$$

   φ ist hierfür undefiniert, allerdings ist die Normalform: $(x+0)+0$

2. Partielle Funktionen: Betrachte hierzu die zusätzliche Regel:

   $$f(s(m), n) \to 0$$

   Dann ist φ undefiniert auf $f(0, 0+0)$, allerdings ist die Normalform: $f(0,0)$

Definition 2.16 *Eine Funktion f heißt **vollständig definiert**, falls ein definierender Baum für f existiert, wobei in jedem Verzweigungsknoten jeder Konstruktor (der entsprechenden Sorte) in einem Teilbaum vorkommt.*

Beispiel: Die Konstruktoren von Nat sind $0, s$. Daher ist $+$ vollständig definiert, die obige Funktion f jedoch nicht.

Für vollständig definierte Funktionen gilt:

Satz 2.9 *Ist R ein induktiv-sequentielles Termersetzungssystem und sind alle Funktionen vollständig definiert, dann ist φ normalisierend auf Grundtermen.*

Diese Eigenschaft ist ausreichend für das Rechnen in funktionalen Sprachen:

- Wir wollen nur Grundterme ausrechnen.
- Falls Normalformen noch definierte Funktionen enthalten, wird dies nicht als Wert angesehen, sondern üblicherweise eine Fehlermeldung ausgegeben.

Vergleichen wir einmal das Pattern matching (Kapitel 1.3) mit der Strategie φ:

Pattern matching:

- abhängig von Reihenfolge der Regeln
- im Allg. nicht normalisierend

- normalisierend auf uniformen Programmen
- erlaubt überlappende linke Regelseiten (dann ist das Ergebnis allerdings von der Regelreihenfolge abhängig!)

Strategie φ:

- unabhängig von der Regelreihenfolge
- normalisierend
- erlaubt keine Überlappungen

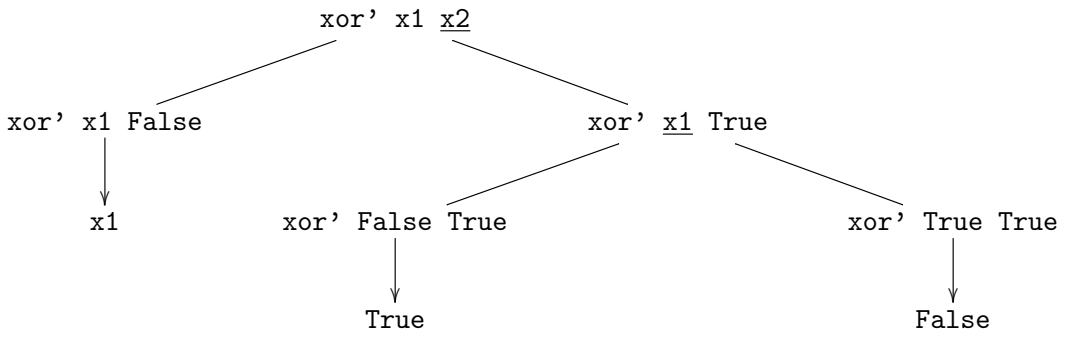
Satz 2.10 *Jede uniforme Funktionsdefinition ist induktiv-sequentiell.*

Die Umkehrung gilt allerdings nicht:

Beispiel: (vgl. Kap. 1.3)

```
xor'  x      False = x
xor'  False  True  = True
xor'  True   True  = False
```

Diese Funktion ist nicht uniform, aber induktiv-sequentiell (durch Verzweigung über das zweite Argument):



Hieraus ist ersichtlich, dass φ berechnungsstärker als einfaches Links-rechts Pattern Matching ist.

Weiterer Aspekt der Strategie φ: Durch eine einfache Erweiterung von definierenden Bäumen auf überlappende Regeln kann man φ zu einer Strategie mit paralleler Auswertung erweitern ([Antoy 92]).

3 Rechnen mit partieller Information: Logikprogrammierung

3.1 Motivation

Prinzipiell sind funktionale Sprachen berechnungsuniversell. Dies bedeutet, dass mit ihnen alles programmierbar ist, was man auch in imperativen Sprachen programmieren kann, und im Prinzip keine Erweiterungen notwendig sind.

Auf der anderen Seite ist jedoch der Programmierstil und die Lesbarkeit von Programmen sehr wichtig. Daher sind Erweiterungen wünschenswert, falls diese zu besser strukturierten oder lesbaren Programmen führen.

Aus diesem Grund betrachten wir noch einmal die wesentliche *Charakteristik funktionaler Sprachen*:

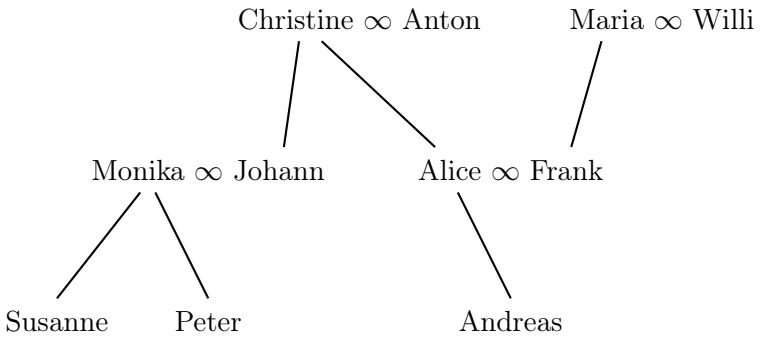
> Variablen stehen für unbekannte Werte, die in Regeln, nicht jedoch in zu berechnenden Ausdrücken vorkommen, d.h. die zu berechnenden Ausdrücke sind immer *Grundterme.*

Eine Lockerung dieser Einschränkung ist durchaus wünschenswert bei *Datenbankanwendungen*, wie das folgende Beispiel zeigt.

Beispiel: Betrachte eine Verwandtschaft, d.h. Personen und Beziehungen zwischen diesen. In der folgenden graphischen Darstellung einer Beispielverwandschaft drücken wir die folgenden Beziehungen aus:

∞ : verheiratet

/ : Mutter-Kind-Beziehung



Das Ziel ist es nun, aus diesen Basisbeziehungen abgeleitete Beziehungen, wie z.B. Tante oder Großvater, allgemein zu definieren.

Zunächst versuchen wir dies in Haskell zu modellieren:

Datentyp der Personen (könnte auch String o.ä. sein):

```
data Person = Christine | Anton | Maria | Willi | Monika | Johann
            | Alice | Frank | Susanne | Peter | Andreas
```

Die „verheiratet"-Beziehung modellieren wir als Funktion `ehemann`:

```
ehemann :: Person  → Person
ehemann   Christine = Anton
ehemann   Maria     = Willi
ehemann   Monika    = Johann
ehemann   Alice     = Frank
```

Die „Mutter-Kind"-Beziehung modellieren wir als Funktion `mutter`:

```
mutter :: Person  → Person
mutter   Johann  = Christine
mutter   Alice   = Christine
mutter   Frank   = Maria
mutter   Susanne = Monika
mutter   Andreas = Alice
```

Aus diesen grundlegenden Beziehungen können wir weitere Beziehungen ableiten, wie z.B.:
Der Vater ist der Ehemann der Mutter (idealerweise!):

```
vater:: Person  → Person
vater kind = ehemann (mutter kind)
```

Allerdings ist die Großvater-Enkel-Beziehung im Allgemeinen eine *Relation*, die so modelliert werden könnte:

```
grossvater :: Person  → Person  → Bool
grossvater g e | g == vater (vater e)  = True
               | g == vater (mutter e) = True
```

Hiermit sind z.B. folgende Berechnungen möglich:

- Wer ist der Vater von Peter?
  `vater Peter` $\leadsto$ `Johann`
- Ist Anton Großvater von Andreas?
  `grossvater Anton Andreas` $\leadsto$ `True`

Leider sind aber keine Berechnungen möglich, bei denen wir andere Werte wissen wollen, wie z.B.:

1. Welche Kinder hat Johann?
2. Welche Großväter hat Andreas?
3. Welche Enkel hat Anton?

Im Prinzip wäre es möglich, diese Fragen auszudrücken, falls *Variablen in zu berechnenden Ausdrücken* zulässig wären:

1. `vater k == Johann` $\leadsto$ `k = Susanne` oder `k = Peter`
2. `grossvater g Andreas` $\leadsto$ `g = Anton` oder `g = Willi`
3. `grossvater Anton e` $\leadsto$ `e = Susanne` oder `e = Peter` oder `e = Andreas`

Die Idee ist also:

> Falls Variablen in Ausdrücken vorkommen, dann suche nach passenden Werten für diese Variablen, sodass die Berechnung des Ausdrucks möglich ist.

Beachte, dass Variablen nicht in den zu berechnenden Ausdrücken funktionaler Sprachen erlaubt sind. In einer funktionalen Sprache müssten wir die Programme zur Erreichung der gleichen Funktionalität erweitern, in dem wir z.B. die Umkehrfunktion/-relation explizit programmieren (z.B. eine Funktion, die zu einem Vater die Menge seiner Kinder berechnet).

In logischen Programmiersprachen, wie z.B. Prolog, ist dagegen die Verwendung von Variablen in Ausdrücken erlaubt. Diese Sprachen basieren auf der Idee, dass *Lösungen* berechnet werden, d.h. Belegungen der Variablen, so dass der Ausdruck reduzierbar ist. Das Hauptproblem dabei ist natürlich, wie man Lösungen konstruktiv findet. Dies werden wir aber erst später erläutern.

Wenn man funktionale Sprachen so erweitert, dass diese auch Aspekte der logischen Programmiersprachen abdecken, dann spricht man auch von **logisch-funktionalen Programmiersprachen**. Bevor wir diese im Detail erläutern, wollen wir die neuen Aspekte, die durch die logische Programmierung ins Spiel kommen, erläutern:

1. **Constraints** (Einschränkungen) statt Gleichheitstests:
   Betrachten wir z.B. die Definition von `grossvater`: Hier ist man nur an positiven Resultaten (d.h. Auswertungen zu `True`) interessiert, aber nicht an Auswertungen, die z.B. besagen, wann jemand kein Großvater einer Person ist. Dies ist im Allgemeinen häufig der Fall. Aus diesem Grund führen wir neben der Gleichheit "`==`", die zu `True` oder `False` auswertet, auch ein **Constraint** der Form

   e_1 `=:=` e_2

ein, wobei e_1 und e_2 auch Variablen enthalten können.

Die Bedeutung eines solchen Constraint ist, dass es *gelöst* werden soll, d.h. es werden Werte für die Variablen in e_1 und e_2 bzw. eine Substitution σ gesucht, so dass $\sigma(e_1)$ und $\sigma(e_2)$ die gleichen *Werte* haben.

Es ist wichtig, den Unterschied zwischen "==" und "=:=" zu verstehen, daher folgen noch einige Anmerkungen dazu:

e_1 `==` e_2 ist ein **Test**: prüfe, ob e_1 und e_2 gleiche oder verschiedene Werte haben. Daher hat "==" den Typ

```
(==) :: a → a → Bool
```

e_1 `=:=` e_2 ist eine **Einschränkung** (**Constraint**): e_1 und e_2 *müssen* gleiche Werte haben (z.B. finde Werte für die Variablen in e_1 und e_2, so dass die Werte dieser Ausdrücke gleich sind). Aus diesem Grund hat der Ausdruck e_1 `=:=` e_2 *nicht* den Typ `Bool`, da z.B. `False` nie ein Ergebnis dieses Ausdrucks ist. Aus diesem Grund hat e_1 `=:=` e_2 hat den speziellen Typ `Success`, d.h.

```
(=:=) :: a → a → Success
```

Der Typ `Success` besitzt dabei keine „wirklichen" Werte, sondern bezeichnet das Ergebnis einer Einschränkung, die erfolgreich sein muss.

Um Einschränkungen miteinander zu größeren Einheiten zu kombinieren, gibt es neben dem **Gleichheitsconstraint** "=:=" noch einige weitere Basisoperationen für Einschränkungen:

```
success :: Success                          -- immer erfüllbar
(&)     :: Success → Success → Success      -- Konjunktion
failed  :: a                                -- Fehlschlag
```

Beispiel: Der Constraint

```
xs++ys =:= [1,2,2] & zs++ys =:= xs
```

kann zu folgenden zwei Antworten erfolgreich ausgewertet werden:

```
xs=[1,2], ys=[2], zs=[1]
xs=[1,2,2], ys=[], zs=[1,2,2]
```

2. **Nichtdeterministische Berechnungen:** Auch bei konfluenten Programmen sind Berechnungen eventuell nichtdeterministisch, da im Allgemeinen mehr als eine Lösung existiert.

3. **Flexiblere Programmierung:** Die Umkehrfunktion/-relationen zu einer definierten Funktionen stehen durch die logischen Anteile automatisch zur Verfügung. Wenn wir z.B. eine Funktion

```
f x = ...
```

definieren, dann können wir diese z.B. in dem Constraint

```
f y =:= e
```

benutzen und erhalten dadurch einen Wert v für die Variable `y`, so dass der Ausdruck `f` v zu (einem Wert von) e auswertet.

4. **Automatische Lösungssuche:** Die Suche nach Lösungen muss nicht explizit programmiert werden, sondern das System sucht automatisch nach passenden Lösungen.

5. **Variablen in initialen Ausdrücken** (auch **Anfragen** genannt) sind existenzquantifiziert, d.h.

```
vater k =:= Johann   where k free
```

bedeutet ∃`k: vater k =:= Johann`. Die explizite `where`-Deklaration der freien Variablen `k` ist notwendig, um dem System mitzuteilen, dass `k` eine freie Variable ist.

Dagegen sind Variablen in Regeln allquantifiziert, d.h.

```
vater k = ehemann (mutter k)
```

bedeutet

∀`k` gilt : `vater k = ehemann (mutter k)`

d.h. hier steht `k` für einen beliebigen Wert.

6. **Variablen in Regeln**:

l `|` c `=` r

Beachte: c kann sowohl vom Typ `Bool` (wie in Haskell) als auch ein Constraint vom Typ `Success` sein!

Übliche Forderung in funktionalen Sprachen: Alle Variablen in c oder r kommen auch in l vor.

In logisch-funktionalen Sprachen können hier, ähnlich wie in initialen Ausdrücken, aber auch **Extravariablen** (Variablen in c oder r, die nicht in l vorkommen) sinnvoll sein. Wir zeigen hierzu einige Beispiele:

- Berechne das letzte Element einer Liste:

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

wobei `ys` und `e` Extravariablen sind und daher explizit deklariert werden müssen.

- Prüfe, ob ein Element in einer Liste enthalten ist:

```
member e xs | ys++(e:zs)=:=xs = success
                              where ys,zs free
```

- Ist eine Liste Teil einer anderen Liste?

```
sublist s l | xs++s++ys=:=l = success
                              where xs,ys free
```

Intuitive Bedeutung von Extravariablen in Regeln:

- Extravariablen sind existenzquantifiziert
- Finde eine Belegung für diese, so dass die Bedingung beweisbar ist.

In Logiksprachen wie Prolog ist dies möglich. Allerdings ist in reinen Logiksprachen die Syntax eingeschränkter, denn dort ist nur die Definition positiver Relationen erlaubt, d.h. die rechte Seite ist immer `success`, so dass jede Gleichung die Form

l `|` c_1 `&` ... `&` c_n `= success`

hat, wobei $c_1, \ldots, c_n$ Literale sind (Anwendung eines Prädikats auf Argumente).

Schreibweise in Prolog:

l `:-` c_1`,...,`c_n`.`

Im folgenden betrachten wir **logisch-funktionale Sprachen**, insbesondere Curry[1]. Dies sind im Prinzip funktionale Sprachen, bei denen aber auch existenzquantifizierte Variablen (auch „freie Variablen“ oder „logische Variablen“ genannt) zulässig sind. Somit können Logiksprachen als Spezialfall logisch-funktionaler Sprachen angesehen werden.

Bevor wir uns mit Auswertungsstrategien logisch-funktionaler Sprachen beschäftigen, zeigen wir noch einige Beispiele zur Programmierung.

Logisch-funktionale Sprachen sind gut geeignet zum Lösen von Suchproblemen, falls man also keinen direkten Algorithmus hat, die Lösung eines Problems zu berechnen.

Beispiel: Spiel 24

Aufgabe:

> Bilde arithmetische Ausdrücke genau mit den Zahlen 2,3,6,8, so dass das Ergebnis der Wert 24 ist (Anmerkung: Division ist ganzzahlig).

[1] `http://www.curry-language.org`

Beispiele für solche Ausdrücke sind: $(3 * 6 - 2) + 8$, $3 * 6 + 8 - 2$,...

Lösungsidee:

1. bilde Pemutationen der Liste `[2,3,6,8]`
2. erzeuge arithmetische Ausdrücke über diesen Permutationen
3. prüfe, ob der Wert 24 ist

Die Berechnung von Permutationen einer Liste kann wie folgt realisiert werden:

```
permute [] = []
permute (x:xs) | u++v =:= permute xs = u++[x]++v where u, v free
```

Beachte, dass `permute` *eine* beliebige Permutation liefert, z.B. wertet `permute [1,2,3]` zu den Werten

```
[1,2,3]
[1,3,2]
[2,1,3]
[2,3,1]
[3,1,2]
[3,2,1]
```

aus. Da diese Operation nichtdeterministisch einen dieser Werte liefert, spricht man auch von einer **nichtdeterministischen Operation**.

Zur Repräsentation von Ausdrücken definieren wir einen Datentyp:

```
data Exp = Num Int      -- Zahl
         | Add Exp Exp
         | Mul Exp Exp
         | Sub Exp Exp
         | Div Exp Exp
```

Nun definieren wir eine Operation, die für einen Ausdruck testet, ob dieser nur Ziffern einer Zahlenliste enthält, und dabei den Wert des Ausdrucks berechnet:

```
test :: Exp  →  [Int]  →  Int
test (Num y)   [z] | y =:= z     = y
test (Add x y) zs  | split u v zs = test x u + test y v   where u, v free
test (Sub x y) zs  | split u v zs = test x u - test y v   where u, v free
test (Mul x y) zs  | split u v zs = test x u * test y v   where u, v free
test (Div x y) zs  | split u v zs = opdiv (test x u) (test y v)
  where
    u, v free
    opdiv a b = if b == 0 || a `mod` b /= 0 then failed else a `div` b
```

Hierbei ist `split` ein Constraint, der erfüllt ist, wenn die ersten beiden Listen nicht-leer sind und zusammen konkateniert gleich der dritten Liste sind:

```
split (u:us) (v:vs) xs | (u:us)++(v:vs) =:= xs = success
```

Nun können wir sehr einfach durch Lösen eines Gleichheitsconstraint Lösungen für unser Problem berechnen:

```
test e (permute [2,3,6,8]) =:= 24
```

Damit erhalten wir folgende Lösungen:

```
e = Add (Sub (Mul (Num 3) (Num 6)) (Num 2)) (Num 8)
e = Add (Mul (Num 3) (Num 6)) (Sub (Num 8) (Num 2))
e = Add (Mul (Num 3) (Sub (Num 8) (Num 2))) (Num 6)
e = Add (Sub (Mul (Num 6) (Num 3)) (Num 2)) (Num 8)
e = Add (Num 6) (Mul (Num 3) (Sub (Num 8) (Num 2)))
e = Add (Mul (Num 6) (Num 3)) (Sub (Num 8) (Num 2))
e = Add (Num 6) (Mul (Sub (Num 8) (Num 2)) (Num 3))
e = Add (Sub (Num 8) (Num 2)) (Mul (Num 3) (Num 6))
e = Add (Mul (Sub (Num 8) (Num 2)) (Num 3)) (Num 6)
e = Add (Num 8) (Sub (Mul (Num 3) (Num 6)) (Num 2))
e = Add (Sub (Num 8) (Num 2)) (Mul (Num 6) (Num 3))
e = Add (Num 8) (Sub (Mul (Num 6) (Num 3)) (Num 2))
e = Sub (Mul (Add (Num 2) (Num 8)) (Num 3)) (Num 6)
e = Sub (Mul (Num 3) (Num 6)) (Sub (Num 2) (Num 8))
e = Sub (Add (Mul (Num 3) (Num 6)) (Num 8)) (Num 2)
e = Sub (Mul (Num 3) (Add (Num 2) (Num 8))) (Num 6)
e = Sub (Mul (Num 3) (Add (Num 8) (Num 2))) (Num 6)
e = Sub (Num 6) (Mul (Num 3) (Sub (Num 2) (Num 8)))
e = Sub (Mul (Num 6) (Num 3)) (Sub (Num 2) (Num 8))
e = Sub (Add (Mul (Num 6) (Num 3)) (Num 8)) (Num 2)
e = Sub (Num 6) (Mul (Sub (Num 2) (Num 8)) (Num 3))
e = Sub (Num 8) (Sub (Num 2) (Mul (Num 3) (Num 6)))
e = Sub (Mul (Add (Num 8) (Num 2)) (Num 3)) (Num 6)
e = Sub (Add (Num 8) (Mul (Num 3) (Num 6))) (Num 2)
e = Sub (Num 8) (Sub (Num 2) (Mul (Num 6) (Num 3)))
e = Sub (Add (Num 8) (Mul (Num 6) (Num 3))) (Num 2)
```

Diese könnte man dann noch natürlich noch schöner ausdrucken.

Nichtdeterministische Operationen

Wir haben gesehen, dass logisch-funktionale Sprachen die Definition nichtdeterministischer Operationen erlauben (wie z.B. `permute`), d.h. Operationen, die zu mehr als einem Wert auswerten. Dies ist in vielen Anwendungen sinnvoll, wie wir oben gesehen haben.

Der Prototyp einer nichtdeterministischen Operation ist der Infixoperator "`?`", der eines seiner Argumente zurück gibt. Dieser ist in Curry vordefiniert durch die Regeln

```
x ? y = x
x ? y = y
```

Falls wir also eine Operation `coin` durch

```
coin = 0 ? 1
```

definieren, wird der Ausdruck `coin` zu 0 oder 1 ausgewertet. Die Verwendung solcher Operationen wird auch in folgendem Beispiel ausgenutzt.

Reguläre Ausdrücke

In diesem Beispiel wollen wir reguläre Ausdrücke zum Pattern Matching verwenden. Hierzu definieren wir zunächst einen Datentyp zur Darstellung regulärer Ausdrücke über einem Alphabet `a`:

```
data RE a = Lit a
          | Alt  (RE a) (RE a)
          | Conc (RE a) (RE a)
          | Star (RE a)
```

Als Beispiel können wir z.B. den Ausdruck $(a|b|c)$ definieren:

```
abc = Alt (Alt (Lit 'a') (Lit 'b')) (Lit 'c')
```

Ein weiteres Beispiel ist der Ausdruck (ab^*):

```
abstar = Conc (Lit 'a') (Star (Lit 'b'))
```

Wir können aber auch einfach die Sprache der regulären Ausdrücke um weitere Konstrukte erweitern, wie z.B. einen `plus`-Operator, der eine mindestens einmalige Wiederholung bezeichnet:

```
plus re = Conc re (Star re)
```

Die Semantik regulärer Ausdrücke kann direkt durch eine Semantik-Funktion definiert werden, wie man dies auch in der Theorie der regulären Ausdrücke macht. Somit wird durch die Semantik jedem regulären Ausdruck ein dadurch beschriebenes Wort zugeordnet. Da es mehrere mögliche Worte geben kann, beschreiben wir dies durch eine nichtdeterministische Operation.

```
sem :: RE a  → [a]
sem (Lit c)    = [c]
sem (Alt  a b) = sem a ? sem b
sem (Conc a b) = sem a ++ sem b
```

```
sem (Star a)   = [] ? sem (Conc a (Star a))
```

Beispielauswertungen:

```
sem abc  ⇝ a oder b oder c
```

Wenn wir einen String gegen einen regulären Ausdruck matchen wollen, können wir dies einfach durch folgendes Constraint beschreiben:

```
match :: RE a  → [a]  → Success
match r s = sem r =:= s
```

Ein Constraint, das ähnlich wie Unix's grep feststellt, ob ein regulärer Ausdruck irgendwo in einem String enthalten ist, können wir wie folgt definieren:

```
grep :: RE a  → [a]  → Success
grep r s = xs ++ sem r ++ ys =:= s  where xs,ys free
```

Beispielauswertungen:

```
grep abstar "dabe"  ⇝ Erfolg!
```

Beachte, dass der letzte Ausdruck eine endliche Auswertung hat, obwohl es unendlich viele Wörter gibt, zu denen `abstar` auswerten kann.

Partielle Datenstrukturen:

In logisch-funktionalen Programmen können Datenstrukturen auch freie Variablen enthalten. Die freien Variablen sind dann Platzhalter für Werte, die evtl. später festgelegt werden

Solche sog. partiellen Datenstrukturen bieten manchmal bei der Programmierung mehr Flexibilität, wie das folgende Beispiel zeigt.

Beispiel: **Maximumbaum**

Gegeben:	Binärbaum mit ganzzahligen Blättern
Gesucht:	Baum mit gleicher Struktur, wobei in jedem Blatt das Maximum aller Blätter des Eingabebaums steht.

Die Definition des Datentyps eines Binärbaums ist einfach:

```
data BTree a = Leaf a | Node (BTree a) (BTree a)
```

Zu implementieren ist eine Funktion `maxTree` mit folgendem Verhalten:

```
maxTree (Node (Leaf 0) (Node (Leaf 1) (Leaf 2)))
⇝ (Node (Leaf 2) (Node (Leaf 2) (Leaf 2)))
```

Eine triviale Lösung besteht aus zwei Baumdurchläufen:

1. Berechnung des Maximums
2. Konstruktion des Maximumbaumes

Mit logischen Variablen ist es dagegen möglich, die Berechnung mit einem Durchlauf zu realisieren, wobei gleichzeitig Baum konstruiert und das Maximum berechnet wird. Dabei enthält der neu konstruierte Baum als Blätter zunächst eine freie Variable an Stelle des noch unbekannten Maximums.

Implementierung:

```
maxTree :: BTree Int → BTree Int
maxTree t  | pass t tmax =:= (newt,tmax)    =  newt
  where
    newt,tmax free

    -- pass berechnet aus einem Baum einen neuen Baum und das Maximum
    -- des Eingabebaums, wobei in den Blättern des neuen Baums
    -- immer das zweite Argument steht
    pass (Leaf n)     mx = (Leaf mx, n)
    pass (Node t1 t2) mx = (Node mt1 mt2, max m1 m2)
           where (mt1, m1) = pass t1 mx
                 (mt2, m2) = pass t2 mx
```

Beachte: Bei der Berechnung von "`pass t tmax`" ist `tmax` zunächst eine freie Variable, die für das erst noch zu berechnende Maximum steht und daher schon in alle Blätter des neuen Baums eingesetzt wird.

Ablauf einer Berechnung:

```
maxTree (Node (Leaf 0) (Leaf 1))
⇝ pass (Node (Leaf 0)(Leaf 1)) tmax =:= (newt,tmax)
   ⇝(Node (Leaf tmax)(Leaf max), 1)
                 newt            tmax
⇝ newt = (Node (Leaf 1) (Leaf 1))
```

3.2 Rechnen mit freien Variablen

In diesem Kapitel wollen wir erläutern, wie man überhaupt mit freien Variablen rechnen kann, insbesondere, wie die Implementierung einer logisch-funktionales Programmiersprache die Bindungen für freie Variablen berechnen kann.

Zu Erinnerung: Die funktionale Programmierung basiert auf dem Prinzip der Termersetzung, d.h. gerechnet wird durch *Reduktion* von Ausdrücken, was im Detail bedeutet:

1. Suche den zu reduzierenden Teilausdruck

2. Suche eine „passende“ Regel (Funktionsgleichung)

3. Ersetze Regelvariablen durch „passende“ Ausdrücke (pattern matching)

Formal ist ein Reduktionsschritt so definiert:

$e \to e[\sigma(r)]_p$

falls $l \to r$ eine Regel ist und σ eine Substitution mit $\sigma(l) = e|_p$.

Beispiel: Die Funktion `f` sei wie folgt definiert:

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

Mittels Reduktion berechenbar:

```
f 0 1 → 0
f 1 2 → 2
f 0 x → 0
```

wobei `x` eine freie Variable ist.

Allerdings können wir mittels Reduktion nicht die Gleichung

```
f z 1 =:= 1
```

lösen, obwohl theoretisch $\sigma = \{\texttt{z} \mapsto 1\}$ eine Lösung wäre, denn

$\sigma($`f z 1 =:= 1`$) \;=\;$ `f 1 1 =:= 1` $\;\to^*\;$ `success`

Um also Ausdrücke, in denen freie Variablen vorkommen, auszurechnen, benötigen wir eine passende Belegung der Variablen, sodass der Ausdruck reduzierbar wird.

Problem: Wie findet man eine passende Belegung?

- Raten: zu ineffizient, da es im Allg. unendlich viele Möglichkeiten gibt.
- Konstruktiv: Ersetze Matching durch Unifikation, was zu dem Begriff **Narrowing** (Verengen des Lösungsraumes) führt.

Im Folgenden betrachten wir Termersetzungssysteme als Programme (da Funktionen höherer Ordnung hier erst einmal nicht so relevant sind), d.h. R ist ein vorgegebenes Termersetzungssystem (was dem eingegebenen Programm entspricht).

Definition 3.1 *Wenn t und t' Terme sind, dann heißt*

$t \rightsquigarrow_\sigma t'$

Narrowingschritt *(bzgl. R), falls gilt:*

1. *p ist eine nichtvariable Position in t (d.h. $t|_p \notin V$).*

2. *$l \to r$ ist Variante einer Regel aus R (d.h. ersetze in der Regel Variablen durch neue Variablen, so dass Namenskonflikte mit freien Variablen vermieden werden).*

3. *σ ist ein mgu (allgemeinster Unifikator, vgl. Kapitel 1.5.2) für $t|_p$ und l, d.h. es gilt $\sigma(t|_p) = \sigma(l)$.*

4. *$t' = \sigma(t[r]_p)$*

Wir führen einige weitere Notationen für Narrowing-Schritte ein:

- *$t \leadsto_{p,l\to r,\sigma} t'$ falls die Position und/oder die Regel wichtig ist.*
- *$t \leadsto_{\sigma'} t'$ mit*

  $$\sigma'(x) = \begin{cases} \sigma(x) & \text{falls } x \in Var(t) \\ x & \text{sonst} \end{cases}$$

  falls die Belegung der Regelvariablen unwichtig ist. σ' ist also die Einschränkung von σ auf die Variablen im Term t (in diesem Fall schreiben wir auch $\sigma' = \sigma|_{Var(t)}$).

- *$t_0 \leadsto^*_\sigma t_n$ falls*

  $$t_0 \leadsto_{\sigma_1} t_1 \leadsto_{\sigma_2} t_2 \leadsto \cdots \leadsto_{\sigma_n} t_n$$

  und $\sigma = \sigma_n \circ \ldots \circ \sigma_1$
  (hierbei bezeichnet $\circ$ die Komposition von Funktionen)

Für das obige Beispiel gibt es zwei mögliche Narrowing-Schritte:

$$\texttt{f z 1} \leadsto_{\{\texttt{z}\mapsto 0\}} \texttt{0}$$

$$\texttt{f z 1} \leadsto_{\{\texttt{z}\mapsto 1\}} \texttt{1}$$

Unterschiede zu Reduktion:

1. Belegung der freien Variablen durch Unifikation
   (beachte: freie Variablen können mehrfach vorkommen, daher wäre in der Bedingung 4. die Forderung $t' = t[\sigma(r)]_p$ **nicht** ausreichend!)
2. Im Allg. gibt es mehrere mögliche Belegungen der Variablen, sodass Narrowing-Schritt **nichtdeterministisch** sind.

Beispiel: Verwandtschaft (vgl. Kap. 3.1)
Betrachten wir die Anfrage: Welche Kinder hat Johann?

```
vater k =:= Johann
→ ehemann (mutter k) =:= Johann
⇝{k↦Susanne} ehemann Monika =:= Johann
→ Johann =:= Johann
→ success
```

Es ist aber auch folgende Berechnung möglich:

```
vater k =:= Johann
→ ehemann (mutter k) =:= Johann
⇝{k↦Peter} ehemann Monika =:= Johann
→ Johann =:= Johann
→ success
```

Die Implementierung von Nichtdeterminismus ist aufwändig, so dass es für eine gute Implementierung wichtig ist, möglichst wenig Nichtdeterminismus zu erzeugen. Hierzu sind gute Strategien notwendig, die wir später erläutern.

Beobachtung: Narrowing ist Verallgemeinerung der Reduktion, was man wie folgt sehen kann:

Wir nehmen an, dass ein Reduktionsschritt möglich ist, d.h. es gilt:

$$t \to t[\sigma(r)]_p$$

mit $\sigma(l) = t|_p$ für eine Regel $l \to r$.
Wir nehmen hierbei an, dass $\mathcal{D}om(\sigma) \subseteq \mathcal{V}ar(l)$ und $\mathcal{V}ar(l) \cap \mathcal{V}ar(t) = \varnothing$ (dies können wir immer erreichen, da $l \to r$ eine Variante eine Regel sein kann). Dann gilt:

$$\sigma(t) = t$$

und somit ist σ ein mgu für l und $t|_p$
Hieraus folgt:

$$t \leadsto_\sigma \underbrace{\sigma(t[r]_p)}_{=t[\sigma(r)]_p}$$

ist ein Narrowing-Schritt.

Somit gilt:

> Jeder Reduktionsschritt ist auch ein Narrowing-Schritt (aber nicht umgekehrt!).

Zusammengefasst ist das Ziel bei der Reduktion die Berechnung einer (bzw. der) Normalform, während das Ziel beim Narrowing das Finden von Lösungen (Variablenbelegungen) ist, sodass eine Normalform berechenbar wird.
Der Ausgangspunkt bei Narrowing-Ableitung sind in der Regel **Gleichungen** (d.h. Constraints), die zu lösen sind. Formal definieren wir dies wie folgt (hier benutzen wir nicht den Gleichheitsoperator "==" von Haskell, da dessen Bedeutung unterschiedlich ist, wie wir noch sehen werden):

Definition 3.2 *Eine **Gleichung** $s \doteq t$ heißt **gültig** (bzgl. R) falls $s \leftrightarrow_R^* t$ (d.h. s kann in t überführt werden).*

Beispiel:
Addition auf natürlichen Zahlen (hierbei bezeichnet s die Nachfolgerfunktion):

$$\begin{array}{rcl} 0+n & \to & n \\ s(m)+n & \to & s(m+n) \end{array}$$

Die Gleichung

$$s(0)+s(0) \doteq s(s(0))$$

ist hier gültig, denn $s(0)+s(0) \to s(0+s(0)) \to s(s(0))$.

Falls R konfluent und terminierend, dann ist $s \leftrightarrow_R^* t$ äquivalent zu der Existenz einer Normalform u mit $s \to^* u$ und $t \to^* u$. Daher reicht es in diesem Fall, die Normalformen beider Seiten zu berechnen, um eine Gleichung zu prüfen.

Mittels Narrowing kann man Gleichungen **lösen**:

Definition 3.3 *Eine Substitution σ heißt **Lösung** der Gleichung $s \doteq t$, falls $\sigma(s) \doteq \sigma(t)$ gültig ist.*

Beispiel: Wir können eine Lösung der Gleichung $z+1=2$ wie folgt berechnen:

$$\begin{array}{lllll} z+s(0) \doteq s(s(0)) & \leadsto_{\{z \mapsto s(m)\}} & s(m+s(0)) & \doteq & s(s(0)) \\ & \leadsto_{\{m \mapsto 0\}} & s(s(0)) & \doteq & s(s(0)) \end{array}$$

Die berechnete Lösung ist dann die Komposition aller Substitutionen eingeschränkt auf die ursprünglichen freie Variablen:

$$\{z \mapsto s(0)\} \quad = \quad (\{z \mapsto s(m)\} \circ \{m \mapsto 0\})|_{\{z\}}$$

Die übliche Forderung an Lösungsverfahren ist:

1. **Korrektheit:** Berechne nur richtige Lösungen
2. **Vollständigkeit:** Berechne alle (bzw. Repräsentanten aller) richtigen Lösungen

Allgemeines Narrowing ist im folgenden Sinn korrekt und vollständig:

Satz 3.1 ([Hullot 80]) *Sei R TES, sodass $\to_R$ konfluent und terminierend ist, und $s \doteq t$ eine Gleichung.*

Korrektheit: *Falls $s \doteq t \leadsto_\sigma^* s' \doteq t'$ und φ ein mgu für s' und t' ist, dann ist $\varphi \circ \sigma$ Lösung von $s \doteq t$, d.h. $\varphi(\sigma(s)) \doteq \varphi(\sigma(t))$ ist gültig.*

Vollständigkeit: *Falls σ' eine Lösung von $s \doteq t$ ist, dann existiert eine Narrowing-Ableitung $s \doteq t \leadsto_\sigma^* s' \doteq t'$, ein mgu φ für s' und t' und eine Substitution τ, so dass $\sigma'(x) \doteq \tau(\varphi(\sigma(x)))$ gültig ist für alle Variablen $x \in \mathcal{V}ar(s \doteq t)$.*

Das Ergebnis zur Vollständigkeit ist deswegen etwas komplizierter formuliert, weil mittels Narrowing nur Repräsentanten von Lösungen, aber nicht alle Lösungen exakt berechnet werden. Betrachten wir dazu das folgende Beispiel:

$$\begin{array}{rcl} f(a) & \to & b \\ g & \to & a \\ i(x) & \to & x \end{array}$$

1. Betrachten wir die Gleichung

   $$f(x) \doteq b$$

   Eine Lösung ist: $\sigma' = \{x \mapsto g\}$
   Narrowing berechnet dagegen:

   $$f(x) \doteq b \leadsto_{\{x \mapsto a\}} b \doteq b$$

   Die berechnete Lösung ist also: $\sigma = \{x \mapsto a\}$
   Es gilt jedoch: $\sigma(x) \doteq \sigma'(x)$ ist gültig.

2. Betrachten wir die Gleichung:

   $$i(z) \doteq z$$

   Eine Lösung ist: $\sigma' = \{x \mapsto a\}$
   Narrowing berechnet nur die eine Lösung: $\sigma = \{\}$ (Identität)
   Es gilt jedoch: σ' ist Spezialfall von σ.

Wir können daher festhalten:

> Narrowing berechnet allgemeine Repräsentanten aller möglichen Lösungen.

Um diese wirklich alle zu berechnen, ist jedoch Folgendes notwendig (wegen der Existenz einer Ableitung):

> Berechne alle möglichen Narrowing-Ableitungen (d.h. rate Position **und** Regel in jedem Schritt!).

Weil dies sehr viele sein können, ist hierzu eine Verbesserung durch spezielle Strategien notwendig ($\leadsto$ später).

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

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

Ist nun

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

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 3.4 *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 == in Haskell, wobei in Haskell keine freien Variablen zugelassen sind. Beachte, dass dies hier erlaubt ist. Wenn z.B. f durch

$$f(x) \to 0$$

definiert ist, dann ist $f(x)$ =:= $f(x)$ gültig!

Interessanter Aspekt: die strikte Gleichheit =:= ist (im Gegensatz zu $\doteq$) durch ein funktionales Programm definierbar:

$$\begin{array}{rcl} c \texttt{ =:= } c & \to & \texttt{success} \quad \text{(für alle 0-stelligen Konstruktoren } c\text{)} \\ d(x_1,\ldots,x_n) \texttt{ =:= } d(y_1,\ldots,y_n) & \to & s_1 \texttt{ =:= } y_1 \texttt{ \& } \ldots \texttt{ \& } s_n \texttt{ =:= } y_n \\ & & \qquad \text{(für alle n-stelligen Konstruktoren } d\text{)} \\ \texttt{success \& } x & \to & x \end{array}$$

Dann gilt:

Satz 3.2 *s =:= t ist gültig $\Leftrightarrow$ s =:= $t \to^*$ `success` mittels obiger Zusatzregeln.*

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

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

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

3.3 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 `Success`, d.h. es sind nur positive Aussagen möglich. Solche Funktionen nennt man auch **Prädikate** oder **Constraints**.
- Alle Regeln haben die Form

  l `= success`

  beziehungsweise

  l `|` c `= success`

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

  Da alle Regeln "`= success`" 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

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

  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 `= success`

  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 folgende Erweiterungen gegenüber funktionalen Sprachen:
  - *Extravariablen* in Bedingung sind 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 unbedingt notwendig wegen der flachen Struktur der Bedingungen.
Bsp.: 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).   % Hierbei ist Ys Zusatzargument für Ergebnis
append([X|Xs], Ys, [X|Zs]) :- append (Xs, Ys, Zs).

rev([],[]).
rev([X|Xs], Ys) :- rev(Xs, Rs), append(Rs, [X], Ys).
% hierbei ist Rs 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 :: $t_1 \to \ldots \to t_n \to$ `Success`

- „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. Betrachte als Beispiel die Definition einer Vorfahrrelation in unserem Verwandtschaftsbeispiel (vgl. Kap. 3.1):

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

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

  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 `success`, 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 success 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 → cond (0=:=0) 0 → cond success 0 → 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 `= success`

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

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

  wobei wir die folgenden Vereinfachungsregeln für `&` verwenden können:

  ```
  success & x = x
  x & success = 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 3.5 (Resolution) *Gegeben sei eine Konjunktion (**Anfrage**, **Ziel**, **goal**) der Form*

l_1 `&...&` l_n

Falls $l_0 = r_0$ (hierbei ist r_0 entweder `success` *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 = success
p b = success
q b = success
```

Anfrage: `p x & q x` $\leadsto_{\{x \mapsto a\}}$ `success & q a` $\leadsto_{\{\}}$ `q a` : Fehlschlag, Sackgasse
$\leadsto_{\{x \mapsto b\}}$ `success & q b` $\leadsto_{\{\}}$ `q b` $\leadsto_{\{\}}$ `success`

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 = success
```

Anfrage: `p x` $\leadsto_{\{x \mapsto a\}}$ `p a` $\to$ `p a` $\to$ `p a` $\to$ ...

Somit haben wir hier eine Endlosschleife, so dass die Lösung $\{x \mapsto b\}$ 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
```

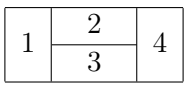
Betrachten wir folgende Anfrage:

```
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

farbe Rot   = success
farbe Gelb  = success
farbe Gruen = success
farbe Blau  = success
```

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 = diff l1 l2 & diff l1 l3 & diff l2 l3 &
                      diff l2 l4 & diff l3 l4

diff x y = (x==y) =:= False
```

   Hier haben wir einen Constraint `diff` definiert, der erfüllt ist, wenn die Argumente verschieden sind.

Eine Lösung können wir dann 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:

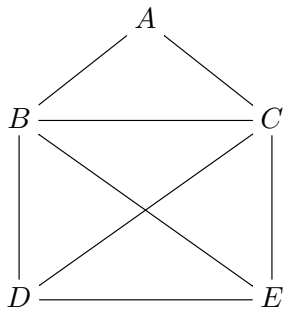
```
faerbung l1 l2 l3 l4 & korrekt l1 l2 l3 l4   where l1,l2,l3,l4 free
↝ l1=Rot, l2=Gelb, l3=Gruen, l4=Rot
↝ l1=Rot, l2=Gelb, l3=Gruen, l4=Blau
⋮
```

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
```

Eine Kante besteht einfach aus zwei Knoten:

```
data Edge = Edge Node Node
```

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 einen Constraint `draw`, der eine Liste von Kanten und Knoten als Argumente hat:

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

Der Constraint "`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 [] [_] = success -- 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 die Eigenschaft von Curry ausgenutzt, dass man auch Muster definieren kann, bei denen Variablen mehrfach vorkommen. Dies ist allerdings nur syntaktischer Zucker für ein Gleichheitsconstraint 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]
↝ nodes=[D,B,A,C,B,E,D,C,E]
⋮
```

3.4 Narrowing-Strategien

Das Reduktionsprinzip aus funktionale Sprachen dient dazu, *Werte* zu berechnen. Durch eine Strategie kann man festlegen, an welcher Stelle Reduktionsschritte in einem Term angewendet werden. Mögliche Strategien sind u.a. die strikte und die lazy Auswertung.

Das Narrowing-Verfahren aus logisch-funktionalen Sprachen wird dagegen verwendet, um *Werten und Lösungen* (d.h. Belegungen freier Variablen) zu berechnen. In diesem Kapitel wollen wir erläutern, welche sinnvollen Strategien es hier gibt.

Beim Narrowing-Verfahren sind wir an der Vollständigkeit interessiert, d.h. wir wollen alle Lösungen bzw. Repräsentanten aller Lösungen finden. Nach dem Satz 3.1 von Hullot (Kap. 3.2) wissen wir, dass wir die Vollständigkeit sicherstellen können, in dem wir *alle Regeln* an *allen passenden Stellen* ausprobieren.

Das Ausprobieren aller Regeln ist im Allg. nicht vermeidbar, wie das folgende Beispiel zeigt:

$$\begin{array}{lcl} f(a) & \to & b \\ f(b) & \to & b \end{array}$$

Betrachten wir nun die Gleichung: $f(x) \doteq b$:

- Durch Anwendung der 1. Regel erhalten wir die Lösung: $\sigma = \{x \mapsto a\}$
- Durch Anwendung der 2. Regel erhalten wir die Lösung: $\sigma = \{x \mapsto b\}$

Diese beiden Lösungen sind unabhängig, sodass jede Regel zu einer Lösung beiträgt.

Ist aber das Ausprobieren aller Positionen in einem Narrowing-Schritt vermeidbar?

- Reduktion: ja ($\leadsto$ lazy evaluation)
- Narrowing: hier ist die Antwort nicht so einfach wegen der Belegung freier Variablen

In diesem Kapitel wollen wir daher Narrowing-Strategien kennenlernen, die das einfache Ausprobieren an allen Positionen verbessern.

3.4.1 Strikte Narrowing-Strategien

Wir betrachten zunächst strikte Narrowing-Strategien. Das Problem hierbei ist allerdings, das die „innerste“ bzw. „äußerste“ Position eventuell nicht eindeutig ist.

Beispiel: Betrachten wir die Regel

$$rev(rev(l)) \to l$$

und den Term

$$rev(rev(x))$$

Welche Position ist die „innerste“? Dies könnte einerseits die Wurzelposition sein, weil wir dort die Regel anwenden können:

$$rev(rev(x)) \leadsto_{1,\ldots,\{\}} x$$

Auf der anderen Seite können wir die Regel auch auf das innere rev anwenden, wenn wir die Variable x passend belegen:

$$rev(rev(x)) \leadsto_{1,\ldots,\{x \mapsto rev(x_1)\}} rev(x_1)$$

Dieses Problem können wir vermeiden, wenn die Termersetzungssysteme *konstrukturbasiert* sind (vgl. Kap.2.2), d.h. wenn bei allen Regeln der Form

$$f(t_1, \ldots, t_n) \to r$$

gilt, dass jedes t_i nur Konstruktoren und Variablen enthält. Die linken Seiten dieser Form nennen wir auch **Muster**.

Nicht zulässig wären dann also die Regeln:

$$\begin{aligned} rev(rev(l)) &\to l \\ (x + y) + z &\to x + (y + z) \end{aligned}$$

Für praktische Programme ist dies keine echte Einschränkung, da funktionale Programme immer konstruktorbasiert sind. Obwohl mit Narrowing auch nicht-konstruktorbasierte Termersetzungssysteme bearbeitet werden, gehören solche Systeme eher in den Bereich der Spezifikation und sind keine effektiv ausführbaren Programme.

Im Folgenden nehmen wir daher an:

Das Termersetzungssystem ist konstruktorbasiert.

Damit können wir nun strikte Narrowing-Strategien definieren:

Definition 3.6 *Eine Narrowing-Ableitung*

$$t_0 \leadsto_{p_1,\sigma_1} t_1 \leadsto_{p_2,\sigma_2} \ldots \leadsto_{p_n,\sigma_n} t_n$$

heißt ***innermost*** *($\approx$ strikt), falls $t_{i-1}|_{p_i}$ ein Muster ist ($i = 1, \ldots, n$).*

Beispiel: Betrachten wir wieder die Additionsoperation:

$$\begin{array}{rcl} 0 + n & \to & n \\ s(m) + n & \to & s(m + n) \end{array}$$

Dann gibt es z.B. folgende Narrowing-Ableitungen:

$x + (y + z) \doteq 0 \quad \leadsto_{1\cdot 2,\{y \mapsto 0\}} \quad x + z \doteq 0 \leadsto_{1,\{x \mapsto 0\}} z \doteq 0$ **innermost**

$x + (y + z) \doteq 0 \quad \leadsto_{1,\{x \mapsto 0\}} \quad y + z \doteq 0 \leadsto_{1,\{y \mapsto 0\}} z \doteq 0$ **nicht innermost**

Innermost Narrowing entspricht der strikten Auswertung in funktionalen Sprachen und ist daher ähnlich effizient implementierbar. Der Nachteil ist allerdings, dass Innermost Narrowing nur eingeschränkt vollständig ist, und zwar aus zwei Gründen:

1. Die berechneten Lösungen sind evtl. zu speziell:

   $$\begin{array}{l} f(x) \to a \\ g(a) \to a \end{array}$$

   Gleichung: $f(g(z)) \doteq a$

   Eine mögliche Lösung ist: $\{\}$ (Identität) (da $f(g(z)) \to_R a$

   Aber: es existiert nur eine Innermost Narrowing-Ableitung:

   $$f(\underline{g(z)}) \doteq a \leadsto_{1\cdot 2,\{z \mapsto a\}} \underline{f(a)} \doteq a \leadsto_{a,\{\}} a \doteq a$$

   Somit berechnet Innermost Narrowing die speziellere Lösung $\{z \mapsto a\}$.

2. Bei partiellen Funktionen schlägt Innermost Narrowing evtl. fehl:

   $$\begin{array}{rcl} f(a, z) & \to & a \\ g(b) & \to & b \end{array}$$

   Gleichung: $f(x, g(x)) \doteq a$
   Innermost Narrowing resultiert in einem Fehlschlag:

   $$f(x, \underline{g(x)}) \doteq s \leadsto_{\{x \mapsto b\}} f(b, b) \doteq a$$

   Dagegen kann das allgemeine Narrowing eine Lösung berechnen:

   $$\underline{f(x, g(x))} \doteq a \leadsto_{\{x \mapsto a\}} a \doteq a$$

Es gibt allerdings eine eingeschränkte Vollständigkeitsaussage von Innermost Narrowing, wie der folgende Satz zeigt:

Satz 3.3 ([Fribourg 85]) *Sei R ein Termersetzungssystem, so dass $\rightarrow_R$ konfluent und terminierend ist. Weiterhin gelte für alle Grundterme t: wenn t in Normalform ist, dann enthält t nur Konstruktoren (d.h. alle Funktionen sind total definiert).*
Dann ist Innermost Narrowing vollständig für alle „Grundlösungen", d.h. es gilt: Falls σ' eine Lösung von $s \doteq t$ ist mit der Eigenschaft, dass $\sigma(x)$ ein Grundterm ist für alle $x \in Var(s \doteq t)$, dann existieren eine Innermost Narrowing-Ableitung $s \doteq t \rightsquigarrow_\sigma^* s' \doteq t'$, ein mgu φ für s' und t' und eine Substitution τ mit $\sigma'(x) \doteq \tau(\varphi(\sigma(x)))$, sodass $\forall x \in Var(s \doteq t)$ gültig ist.

Viele Funktionen sind total definiert, so dass Innermost Narrowing dafür vollständig ist. Falls allerdings auch partielle Funktionen vorhanden sind, dann kann man Innermost Narrowing so erweitern, dass man bei partiellen Funktionen nicht fehlschlägt. Diese Erweiterung heißt **Innermost Basic Narrowing** und basiert auf folgender Idee:

- Erlaube das „Überspringen" innerer Funktionsaufrufe
- Falls ein Aufruf übersprungen wurde, ignoriere diesen auch in allen nachfolgenden Schritten (diese Technik wurde auch unabhängig vom Innermost Narrowing als **Basic Narrowing** in [Hullot 80] vorgeschlagen).

Für eine genaue formale Beschreibung ist es notwendig, zwischen auszuwertenden und übersprungenen Funktionsaufrufen zu unterscheiden. Hierzu stellen wir Gleichungen als Paar

$$\langle E;\ \sigma \rangle$$

dar, wobei das „Skelett" E die auszuwertenden und die „Umgebung" σ die übersprungenen Funktionsaufrufe enthält. Das Paar $\langle E;\ \sigma \rangle$ steht dabei für den Ausdruck $\sigma(E)$.

Intuitiv wird diese Paardarstellung wie folgt verwendet:

- Auswertung: hier werden nur Aufrufe in E ausgewertet
- „Überspringen": verschiebe einen Aufruf von E nach σ
- Eine initiale Gleichung E wird als Paar $\langle E; [] \rangle$ dargestellt.

Beispiel: Betrachten wir das oben angegebene Termersetzungssystem R mit einer partiellen Funktion:

$$\begin{array}{rcl} f(a,z) & \rightarrow & a \\ g(b) & \rightarrow & b \end{array}$$

Gleichung: $f(x, g(x)) \doteq a$

Darstellung als Paar:

$$\begin{array}{lll} & \langle f(x, g(x)) \doteq a; \{\}\rangle & \\ \leadsto & \langle f(x, y) \doteq a; \{y \mapsto g(x)\}\rangle & \text{„überspringen“} \\ \leadsto_{\{x \mapsto a\}} & \langle a \doteq a; \{y \mapsto g(a), x \mapsto a\}\rangle & \text{„Innermost Narrowing im Skelett“} \end{array}$$

Damit haben wir die Lösung: $\{x \mapsto a\}$ berechnet, d.h. mittels des “Überspringens“ ist Innermost Basic Narrowing vollständig.

Wir wollen nun diese Strategie formal beschreiben: Innermost basic Narrowing besteht aus zwei alternativen Ableitungsschritten:

Narrowing:

- $p \in \mathcal{P}os(E)$ mit $E|_p$ ist ein Muster
- $l \to r$ ist eine neue Variante einer Regel
- σ' ist ein mgu für $\sigma(E|_p)$ und l

Dann ist

$$\langle E; \sigma\rangle \ \leadsto \ \langle E[r]_p; \sigma' \circ \sigma\rangle$$

ein „Innermost Basic Narrowing-Schritt“

Innermost reflection:

- $p \in Pos(E)$ mit $E|_p$ ist ein Muster
- x ist eine neue Variable mit $\sigma' = \{x \mapsto \sigma(E|_p)\}$

Dann ist

$$\langle E; \sigma\rangle \ \leadsto \ \langle E[x]_p; \sigma' \circ \sigma\rangle$$

ein „Überspringen“-Schritt.

Anmerkungen:

- In beiden Fällen kann p auch die linkeste Position sein.
- Innermost basic Narrowing ist vollständig für konfluente und terminierende Termersetzungssysteme.

Reine innermost Narrowing-Verfahren haben trotz dieser Verbesserung einen Nachteil (im Unterschied zu innermost Reduktionstrategien!): Sie laufen häufig in Endlosschleifen, wenn rekursive Datenstrukturen verwendet werden. Dies passiert auch, wenn das Termersetzungssystem terminierend ist!

Beispiel:

$$\begin{array}{rcl} 0 + n & \to & n \\ s(m) + n & \to & s(m + n) \end{array}$$

Dieses Termersetzungssystem ist konfluent, terminierend und total definiert, d.h. es hat alle für Innermost Narrowing wünschenswerten Eigenschaften. Allerdings gibt es schon bei einfachen Gleichungen endlose Innermost Narrowing-Ableitungen:

$$x + y \doteq 0 \leadsto_{\{x \mapsto s(x_1)\}} s(x_1 + y) \doteq 0 \leadsto_{\{x_1 \mapsto s(x_2)\}} s(s(x_2 + y)) \doteq 0 \leadsto \ldots$$

Der letzte und alle weiteren Narrowing-Schritte sind überflüssig, da die linke und rechte Seite der Gleichung nie gleich werden kann, da s und 0 verschiedene Konstruktoren sind.

Eine Verbesserung kann erreicht werden, wenn wir prüfen (vor einem Narrowing-Schritt!), ob bei einer Gleichung verschiedene Konstruktoren außen stehen. Hierzu definieren wir:

$$\mathcal{H}ead(t) = f \quad :\Leftrightarrow \quad t = f(t_1, \ldots, t_n)$$

Rückweisung: („rejection"):
Falls $s \doteq t$ die zu lösende Gleichung ist und $p \in Pos(s) \cup Pos(t)$ mit $Head(s|_p) \neq Head(t|_p)$ und $Head(s|_{p'}), Head(t|_{p'}) \in C$ für alle Positionen $p' \leq p$ (hierbei ist C wie immer die Menge der Konstruktoren), dann hat $s \doteq t$ keine Lösung, d.h. wir können die Narrowing-Ableitung sofort mit einem Fehlschlag abbrechen.

Allerdings gibt es auch mit der Rückweisungsregel noch viele überflüssige Ableitungen:

$$\begin{array}{lll} \underline{(x+y)} + z \doteq 0 & \leadsto_{\{x \mapsto s(s_1)\}} & s(\underline{x_1 + y}) + z \doteq 0 \\ & \leadsto_{\{x_1 \mapsto s(x_2)\}} & s(s(\underline{x_2 + y})) + z \doteq 0 \\ & \leadsto \ldots & \end{array}$$

Bei dieser Ableitung erfolgt keine Rückweisung, da + eine definierte Funktion und kein Konstruktor ist!

Eine weitere Verbesserung können wir dadurch erreichen, dass wir *vor* jedem Narrowing-Schritt den Term zur Normalform reduzieren. Dieses erweiterte Verfahren wird auch als **normalisierendes Innermost Narrowing** bezeichnet [Fay 79, Fribourg 85].

Betrachten wir erneut das obige Beispiel und führen es mit normalisierendem Innermost Narrowing aus:

$$\begin{array}{lll} \underbrace{(x+y) + z \doteq 0}_{in\ Normalform} & \leadsto_{\{x \mapsto s(x_1)\}} & s(x_1 + y) + z \doteq 0 \\ & \rightarrow & s((x_1 + y) + z) \doteq 0 \\ & & \text{Rückweisung: Fehlschlag!} \end{array}$$

Normalisierendes Innermost Narrowing hat einige interessante Eigenschaften:

- Normalisierung ist ein deterministischer Prozess (für konfluente Termersetzungssysteme), da wir der Normalisierung Priorität geben (diese findet ja *vor* dem Narrowing statt), führt dies zu einer kürzeren Laufzeit und geringerem Speicherplatzverbrauch.

Beispiel:

$$\begin{array}{lcl} 0 * n & \rightarrow & 0 \\ n * 0 & \rightarrow & 0 \end{array}$$

Gleichung: $x * 0 \doteq 0$
Innermost Narrowing erlaubt zwei verschieden Ableitungen:

$$\begin{array}{lll} x * 0 \doteq 0 & \leadsto_{\{x \mapsto 0\}} & 0 \doteq 0 \\ x * 0 \doteq 0 & \leadsto_{\{\}} & 0 \doteq 0 \end{array}$$

Innermost Narrowing mit Normalisierung ist dagegen deterministisch, weil nur eine Ableitung möglich ist:

$$x * 0 \doteq 0 \rightarrow 0 \doteq 0$$

- Vermeidung unnötiger Ableitungen
- Durch diese Technik können logisch-funktionale Programme effizienter abgearbeitet werden als rein logische Programme (s.u.).
- Es ist effizient implementierbar, indem man ähnliche Implementierungstechniken wie für funktionale oder logische Programmiersprachen verwendet (vgl. [Hanus 90, Hanus 91, Hanus 92])

Nachteil rein logischer Programme:

- Sie haben eine flache Struktur.
- Sie haben keine expliziten funktionalen Abhängigkeiten.

Beispiel: Betrachten wir die Implementierung des obigen Beispiels in Prolog:

```
add(0, N, N).  % hier ist N ein Extraargument für das Funktionsergebnis
add(s(M), N, s(z)) :- add(M, N, Z).
```

Die Anfrage ist:

```
add(X, Y, R), add (R, Z, 0)
```

Hierbei ist `R` eine Hilfsvariable, die notwendig ist zur Abflachung des Terms `((x*y)+z)`. Zu dieser Anfrage gibt es die folgende endlose Ableitung:

`add(X, Y, R), add (R, Z, 0)`
$\leadsto_{\{X \mapsto s(X1), R \mapsto s(R1)\}}$ `add(X1, Y, R1), add(s(R1), Z, 0)`
$\leadsto_{\{X1 \mapsto s(X2), R1 \mapsto s(R1)\}}$ `add(X2, Y, R2), add(s(s(R2)), Z, 0)`
$\leadsto \cdots$

Die Voraussetzung für alle Innermost-Strategien und auch für die Normalisierung ist die Terminierung des Termersetzungssystems. Diese Voraussetzung ist allerdings sehr streng, denn

- sie ist schwer zu prüfen (unentscheidbar!)
- sie verbietet bestimmte funktionale Programmiertechniken (unendliche Datenstrukturen, Kap. 1.6)

Bei den Reduktionsstrategien hatten wir gesehen, dass outermost oder lazy Strategien bei nichtterminierenden Termersetzungssystemen vorteilhaft sind. Aus diesem Grund wollen wir nachfolgend diskutieren, ob dies auch bei Narrowing-Strategien so ist.

3.4.2 Lazy Narrowing-Strategien

Strikte Strategien haben die generelle Eigenschaft, dass alle Teilausdrücke ausgewertet werden. Wie wir wissen, hat dies den potenziellen Nachteil, dass strikte Strategien

- bei nichtterminierenden Funktionen unvollständig
- im Allgemeinen nicht optimal

sind.

Dagegen wertet eine lazy Strategie Teilterme nur dann aus, falls es notwendig ist, sodass man damit ein potenziell besseres Verhalten erzielen kann. Was bedeutet allerdings „falls es notwendig ist“?
Eine präzise formale Definition ist schon bei Reduktionsstrategien nicht einfach und bei Narrowing-Strategien, wo wir noch die Belegung freier Variablen berücksichtigen müssen, noch schwieriger zu definieren. Aus diesem Grund versuchen wir als ersten Ansatz eine einfache Definition, indem wir „lazy“ durch „outermost“ ersetzen.

Definition 3.7 *Ein Narrowing-Schritt $t \leadsto_{p,\sigma} t'$ heißt **outermost**, falls für alle Narrowing-Schritte $t \leadsto_{p',\sigma'} t''$ gilt: $p' \not< p$ (p' nicht über p). In analoger Weise können wir auch **leftmost outermost** definieren als: $p' \not< p$ und p' nicht links von p.*

Leider zeigt sich, dass Outermost Narrowing im Allgemeinen unvollständig ist (auch wenn es nur eine outermost Position gibt, was im Gegensatz zur Reduktion steht!).

Beispiel: Betrachten wir das folgende Termersetzungssystem:

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

Dieses Termersetzungssystem ist konfluent, terminierend und total definiert. Betrachten wir nun die Gleichung:

$$f(f(i,j),k) \doteq 0$$

Mittels Innermost Narrowing kann die folgende Lösung berechnet werden:

$$\begin{array}{lll} f(f(i,j),k) \doteq 0 & \leadsto_{1\cdot 1,\{i\mapsto 0, j\mapsto 0\}} & f(0,k) \doteq 0 \\ & \leadsto_{1,\{k\mapsto 0\}} & 0 \doteq 0 \end{array}$$

Die hierbei berechnete Lösung ist: $\{i \mapsto 0, j \mapsto 0, k \mapsto 0\}$

Mittels Outermost Narrowing wird allerdings *keine* Lösung berechnet, da hier nur ein Outermost Narrowing-Schritt möglich ist:

$$f(f(i,j),k) \doteq 0 \quad \leadsto_{1,\{k\mapsto s(y)\}} \quad s(0) \doteq 0$$

Um die Vollständigkeit von Outermost Narrowing sicherzustellen sind weitere Restriktionen notwendig [Echahed 88, Padawitz 87]. Informell ist **Outermost Narrowing vollständig** (im Sinn von Satz 3.1 von Hullot), falls das Termersetzungssystem konfluent und terminierend ist und jeder Narrowing-Schritt $t \leadsto_{p,\sigma} t'$ (bzgl. dieser Strategie) **uniform** ist. Letzteres bedeutet, dass Narrowing-Positionen invariant unter Normalforminstanziierung sind, d.h. $\forall \varphi$ mit $\varphi(x)$ ist in Normalform $\forall x \in \mathcal{D}om(\varphi)$ existiert ein Narrowing-Schritt $\varphi(t) \leadsto_{p,\sigma'} t''$.

Beispiel: Outermost Narrowing ist nicht uniform, da z.B.

$$f(f(i,j),k) \doteq 0 \leadsto_{1\cdot 1,\{k\mapsto s(y)\}} 0 \doteq 0$$

ein Narrowing-Schritt ist, aber ein Narrowing-Schritt für

$$f(f(i,j),0) \doteq 0 \leadsto_{1\cdot 1,\ldots}$$

an der gleichen Position nicht möglich ist.

Dieses theoretische Vollständigkeitskriterium hat einige Nachteile:

- Uniformität ist schwer zu prüfen.
- Uniformität ist eine harte Einschränkung (dies ist eine stärkere Einschränkung als total definiert!).
- Die immer noch geforderte Terminierungseigenschaft verhindert bestimmte funktionale Programmiertechniken.

Aus diesem Grund betrachten wir nun Strategien, bei denen auf die Terminierungsforderung verzichtet wird. Stattdessen fordern wir, dass das Termersetzungssystem konstruktorbasiert und schwach orthogonal ist, was wir auch mit **KB-SO** abkürzen.

Zu beachten ist, dass bei nichtterminierenden Termersetzungssystemen die reflexive Gleichheit $\doteq$ nicht sinnvoll ist (vgl. Kapitel 3.2), sodass wir nachfolgend immer die strikte Gleichheit `=:=` betrachten (vgl. Definition 3.4).

Definition 3.8 (Lazy Narrowing (informell)) *Eine Narrowing-Position ist **lazy**, falls dies die Wurzel ist oder der Wert an dieser Stelle notwendig ist, um eine Regel an einer darüberliegenden Position anzuwenden (die entsprechende formale Definition kann man in [Moreno-Navarro/Rodríguez-Artalejo 92] finden).*

Beispiel: Wir betrachten noch einmal das obige Termersetzungssystem

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

und den Term

$$f(f(i,j),k)$$

1. Das äußere Vorkommen von f ist eine lazy Narrowing-Position, weil dies die Wurzelposition des Terms ist.

2. Das innere Vorkommen von f ist auch eine lazy Narrowing-Position, weil diese notwendig ist, um die erste oder zweite Regel an der Wurzelposition anzuwenden.

Definition 3.9 *Ein Narrowing-Schritt $t \leadsto_{p,\sigma} t'$ heißt **lazy**, falls p eine lazy Position in t ist.*

Satz 3.4 ([Moreno-Navarro/Rodríguez-Artalejo 92]) *Lazy Narrowing ist vollständig für KB-SO Termersetzungssysteme und strikte Gleichungen.*

Somit haben wir damit eine Narrowing-Strategie, die das Lösen von Gleichungen mit unendlichen Strukturen ermöglicht. Betrachten wir hierzu als Beispiel das folgende Termersetzungssystem:

$$\begin{array}{rcl} from(n) & \to & n : from(s(n)) \\ first(0,xs) & \to & [] \\ first(s(n),x:xs) & \to & x : first(n,xs) \end{array}$$

Dann können wir die folgende Gleichung über potenziell unendlichen Strukturen lösen:

$$\begin{array}{ll} & first(x,\ \underline{from(y)}) \texttt{ =:= } [0] \\ \leadsto_{\{\}} & \underline{first}(x,\ y : from(s(y))) \texttt{ =:= } [0] \\ \leadsto_{\{x \mapsto s(x_1)\}} & y : first(x_1,\ from(s(y))) \ \underline{\texttt{=:=}}\ 0 : [] \\ \leadsto_{\{\}} & y \underline{\texttt{=:=}} 0 \ \&\ first(x_1,\ from(s(y))) \texttt{ =:= } [] \\ \leadsto_{\{y \mapsto 0\}} & success \ \underline{\&}\ first(x_1,\ from(s(y))) \texttt{ =:= } [] \\ \leadsto_{\{\}} & \underline{first}(x_1,\ from(s(y))) \texttt{ =:= } [] \\ \leadsto_{\{x_1 \mapsto 0\}} & [] \texttt{ =:= } [] \\ \leadsto_{\{\}} & success \end{array}$$

Somit ist die berechnete Lösung: $\{x \mapsto s(0), y \mapsto 0\}$

Dagegen laufen alle strikten Strategien hier in eine Endlosschleife!

Im Gegensatz zur Reduktion in funktionalen Sprachen gibt es für einen Term t nicht nur eine lazy Position, sondern eventuell mehrere lazy Narrowing-Positionen. Dies kann tatsächlich zu dem Problem führen, dass sich manche lazy Narrowing-Schritte als überflüssig bzgl. bestimmter Substitutionen herausstellen, was auf Grund der Interaktion von Variablenbindung und Positionsauswahl passieren kann.

Betrachten wir hierzu das folgende Termersetzungssystem:

$$\begin{array}{rcl} 0 \leq n & \to & True \\ s(m) \leq 0 & \to & False \\ s(m) \leq s(n) & \to & m \leq n \end{array} \qquad \begin{array}{rcl} 0 + n & \to & n \\ s(m) + n & \to & s(m + n) \end{array}$$

Der auszuwertende Ausdruck sei

$$x \leq y + z$$

Dieser Ausdruck hat zwei lazy Narrowing-Positionen: der gesamte Ausdruck bzw. das zweite Argument.

Eine Lazy Narrowing-Ableitung ist daher z.B.

$$x \leq y + z \quad \leadsto_{\{x \mapsto 0\}} \quad True$$

Eine weitere Lazy Narrowing-Ableitung ist z.B.

$$x \leq y + z \quad \leadsto_{\{y \mapsto 0\}} \quad x \leq z \quad \leadsto_{\{x \mapsto s(x_1), z \mapsto 0\}} \quad False$$

Jedoch ist auch dies eine Lazy Narrowing-Ableitung:

$$x \leq y + z \quad \leadsto_{\{y \mapsto 0\}} \quad x \leq z \quad \leadsto_{\{x \mapsto 0\}} \quad True$$

Vergleichen wir nun die erste und die dritte Ableitung:

1. Ableitung: berechnete Lösung: $\{x \mapsto 0\}$, Schritte: 1

3. Ableitung: berechnete Lösung: $\{x \mapsto 0, y \mapsto 0\}$, Schritte: 2

Die 3. Lazy Narrowing-Ableitung enthält somit einen überflüssigen Schritt und berechnet eine zu spezielle Lösung. Somit können wir festhalten:

Lazy Narrowing ist nicht wirklich lazy!

Den „Fehler“, der in der 3. Ableitung gemacht wurde, können wir wie folgt erklären:

> Das Argument $y + z$ wurde ausgewertet, weil die 2. oder 3. Regel für $\leq$ später angewendet werden sollte, d.h. x sollte später eigentlich an $s(x_1)$ (und nicht an 0) gebunden werden. Trotzdem haben wir nachher (weil dies ja ein zulässiger Lazy Narrowing-Schritt ist) x an 0 gebunden und die 1. Regel angewendet.

Wir können daher dieses Problem nur vermeiden, indem wir x frühzeitig an den beabsichtigten Wert binden. Somit wäre eine mögliche Strategie:

- Binde x an 0 oder $s(x_1)$.
- Wende, je nach Bindung, entweder die 1. $\leq$-Regel an (falls x an 0 gebunden wurde) oder werte das Argument $y + z$ aus (falls x an $s(x_1)$ gebunden wurde).

Diese verbesserte Strategie wird auch **Needed Narrowing** genannt [Antoy/Echahed/Hanus 00]. Die Grundidee bei der Auswertung des Aufrufs $f(t_1, \ldots, t_n)$ ist hierbei:

1. Bestimme ein Argument t_1, dessen Wert von allen Regeln für f verlangt wird (bei der Funktion $\leq$ ist dies das 1. Argument, nicht jedoch das 2. Argument).
2. Falls t_1
   - ein Konstruktor $c(\ldots)$ ist: wähle die passende Regel mit diesem Konstruktor
   - ein Funktionsaufruf $g(\ldots)$ ist: werte diesen aus
   - eine Variable ist: binde diese (nichtdeterministisch) an die verschiedene Konstruktoren und mache weiter

Beispiel: Da bei der Funktion $\leq$ der Wert des ersten Arguments verlangt wird, gibt es die folgenden Needed Narrowing-Ableitungen:

$$x \leq y+z \overset{Binde}{\longmapsto}_{\{x\mapsto 0\}} 0 \leq y+z \to_R True$$

$$\overset{Binde}{\longmapsto}_{\{x\mapsto s(x_1)\}} s(x_1) \leq \underbrace{y+z}_{*} \overset{Binde}{\longmapsto}_{\{y\mapsto 0\}} s(x_1) \leq 0+z \to_R s(x_1) \leq z$$

$$\overset{Binde}{\longmapsto}_{\{z\mapsto 0\}} s(x_1) \leq 0 \to_R False$$

$$\overset{Binde}{\longmapsto}_{\{z\mapsto s\mapsto z_1)\}} \cdots$$

$$\overset{Binde}{\longmapsto}_{\{y\mapsto s(y_1)\}} \cdots$$

$*$ = auswerten, dabei y verlangt

Man beachte, dass die obige 3. Ableitung hier nicht möglich ist! Allerdings ist der Schritt

$$x \leq y+z \rightsquigarrow_{\{x\mapsto s(x_1), y\mapsto 0\}} s(x_1) \leq z$$

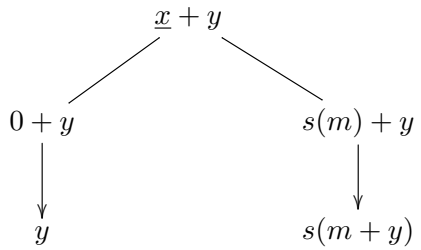
kein Narrowing-Schritt im bisherigen Sinn, da $\{x \mapsto s(x_1), y \mapsto 0\}$ kein **mgu** für den Teilterm $y+z$ und die linke Regelseite ist (sondern dies ist nur ein Unifikator)! Wir sehen also, dass die Berechnung speziellerer Unifikatoren für eine wirkliche „lazy“ Strategie essentiell ist!

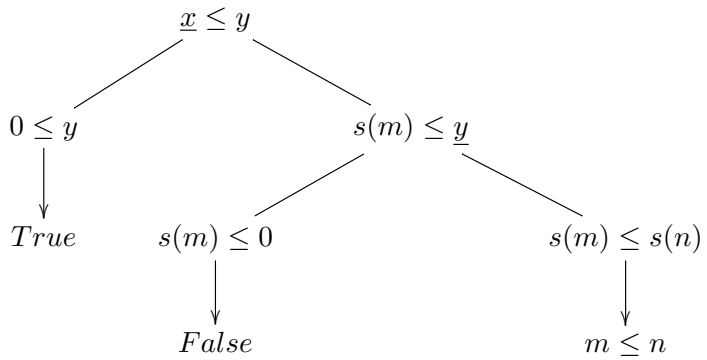
Das Problem, was sich stellt, ist:

> Wie findet man die „verlangten“ Argumente?

Generell ist dies ein unentscheidbares Problem für orthogonale Termersetzungssysteme. Allerdings ist dies für induktiv-sequentielle Systeme (vgl. Kapitel 2.2) einfach realisierbar mittels definierender Bäume.

Betrachten wir hierzu die definierenden Bäume für die Funktionen + und $\leq$:





Zum Beispiel legt der definierende Baum für $\leq$ die folgende Strategie zur Auswertung von $t_1 \leq t_2$ fest: Da das 1. Argument t_1 verlangt ist, mache eine Fallunterscheidung über dieses Argument:

1. Falls t_1 ein Funktionsaufruf ist, werte diesen aus und fahre dann mit der gleichen Strategie fort.
2. Falls $t_1 = 0$: wende Regel an
3. Falls $t_1 = s(\ldots)$: Mache eine Fallunterscheidung über das 2. Argument t_2:
   - Falls t_2 ein Funktionsaufruf ist: werte ihn aus
   - Falls $t_2 = 0$ oder $= s(\ldots)$: wende die entsprechende Regel an
   - Falls t_2 eine Variable ist: Binde t_2 an 0 oder $s(x)$ (wobei x eine neue Variable ist) und wende die entsprechende Regel an.
4. Falls t_1 eine Variable ist: Binde t_1 an 0 oder $s(x)$ (wobei x eine neue Variable ist) und mache mit 2. oder 3. weiter.

Formal können wir die Needed Narrowing-Strategie wie folgt definieren. Hierbei ist R ein induktiv-sequentielles Termersetzungssystem.

Definition 3.10 (Needed Narrowing-Schritt) *Sei s ein Term, o eine Position des linkesten äußersten definierten Funktionssysmbols in s (d.h. $s|_o = f(t_1, \ldots, t_n)$) und $\mathcal{T}_f$ ein definierender Baum für f. Die* ***Needed Narrowing-Strategie*** *λ berechnet Tripel der Form $(p, l \to r, \sigma)$, so dass $s \leadsto_{p,l\to r,\sigma} \sigma(s[r]_p)$ ein Narrowing-Schritt ist (beachte allerdings, dass σ nicht unbedingt ein **m**gu ist). λ ist hierbei wie folgt definiert:*

$$\lambda(s) = \{o \cdot p, l \to r, \sigma) \mid (p, l \to r, \sigma) \in \lambda(s|_o, \mathcal{T}_f)\}$$

wobei $\lambda(t, \mathcal{T})$ die kleinste Tripelmenge mit

$$\lambda(t, \mathcal{T}) \supseteq \begin{cases} \{(\epsilon, l \to r, mgu(t, l))\} & \text{falls } \mathcal{T} = l \to r \\ \lambda(t, \mathcal{T}_i) & \text{falls } \mathcal{T} = branch(\pi, p, \mathcal{T}_1, \ldots, \mathcal{T}_n) \\ & \text{und } t \text{ und } pattern(\mathcal{T}_i) \text{ unifizierbar} \\ \{(p \cdot p', l \to r, \sigma \circ \tau)\} & \text{falls } \mathcal{T} = branch(\pi, p, \mathcal{T}_1, \ldots, \mathcal{T}_n), \\ & t|_p = g(\ldots) \text{ mit } g \in D,\ \tau = mgu(t, \pi) \\ & \mathcal{T}_g \text{ def. Baum für } g \text{ und } (p', l \to r, \sigma) \in \lambda(\tau(t|_p), \mathcal{T}_g) \end{cases}$$

ist.

Anmerkung: Bei der letzten Alternative ist die Anwendung von τ auf den Teilterm $t|_p$ notwendig wegen eventueller Mehrfachvorkommen von Variablen, wie das folgende Beispiel zeigt:

$$\underline{x} \leq x + x \;\longmapsto_{\{x \mapsto s(x_1)\}}\; s(x_1) \leq s(x_1) + s(x_1) \;\rightarrow\; s(x_1) \leq s(x_1 + s(x_1))$$

Hier ist also

$$(2, s(m_1) + n \rightarrow s(m_1 + n), \{x \mapsto s(x_1), m_1 \mapsto x_1, n \mapsto s(x_1)\}) \in \lambda(x \leq x + x)$$

Diese formale Definition ist zwar etwas unhandlich, aber tatsächlich ist Needed Narrowing einfach implementierbar analog zum Pattern Matching (denn branch-Knoten entsprechen case-Ausdrücken).

Satz 3.5 ([Antoy/Echahed/Hanus 00]) *Needed Narrowing ist vollständig für induktiv-sequentielle Termersetzungssystem bzgl. strikter Gleichungen.*

Needed Narrowing hat darüber hinaus weitere wichtige Eigenschaften:

1. **Optimalität:**
   - Jeder Schritt ist notwendig („needed“), d.h. falls eine Lösung berechnet wird, war kein Schritt in dieser Berechnung überflüssig.
   - Needed Narrowing-Ableitungen haben minimale Länge (dies gilt allerdings nur, falls identische Terme nur einmal berechnet werden, wie dies mittels “sharing” auch bei nicht-strikten funktionalen Sprachen gemacht wird).
   - Minimale Lösungsmenge: Falls σ und σ' Lösungen sind, die durch verschiedene Ableitungen berechnet werden, dann sind σ und σ unabhängig (d.h. $\sigma(x)$ und $\sigma'(x)$ sind nicht unifizierbar für eine Variable x).
2. **Determinismus:**
   Bei Auswertung variablenfreier Terme ist jeder Schritt deterministisch. In der Konsequenz bedeutet dies, dass funktionale Programme deterministisch ausgewertet werden und Nichtdeterminismus nur bei Vorkommen freier Variablen auftritt.

Erweiterung von Needed Narrowing

Needed Narrowing ist nur für induktiv-sequentielle Termersetzungssysteme definiert. Dies bedeutet, dass *überlappende Regeln* bei Needed Narrowing nicht erlaubt sind. Als Beispiel betrachten wir hierzu noch einmal das „parallele Oder“:

```
R: True  ∨ x    → True
   x     ∨ True → True
   False ∨ False → False
```

Das Problem bei dieser Definition ist, dass es kein eindeutiges Induktionsargument gibt, d.h. kein Argument ist "needed", sodass bei der Auswertung von $t_1 \vee t_2$ unklar ist, ob man zuerst t_1 oder zuerst t_2 auswerten soll.

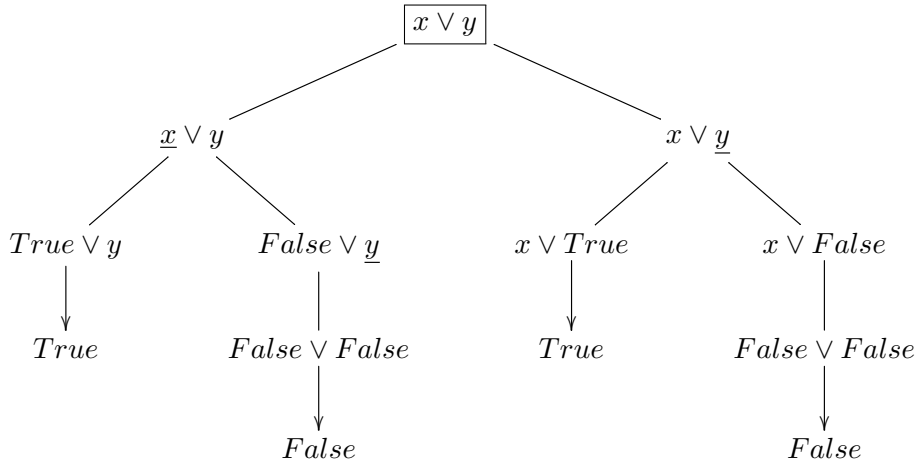
Weil allerdings solche überlappenden Definitionen in der Praxis durchaus vorkommen können, erweitern wir definierende Bäume, indem wir zusätzlich Oder-Knoten, d.h. Knoten für alternative Auswertungen, einführen:

Definition 3.11 *Ein **erweiterter definierender Baum** ist ein definierender Baum, der folgende Knoten mit einem Muster π enthalten kann:*

- *Regelknoten: wie bisher*
- *Verzweigungsknoten: wie bisher*
- ***Oder-Knoten** der Form $or(\mathcal{T}_1, \ldots, \mathcal{T}_n)$, wobei jedes $\mathcal{T}_i$ ein erweiterter definierender Baum mit dem Muster π ist.*

$\mathcal{T}$ heißt erweiterter definierender Baum für eine n-stellige Funktion f, falls $\mathcal{T}$ endlich ist und das Muster $f(x_1, \ldots, x_n)$ hat ($x_1, \ldots, x_n$ sind verschiedene Variablen), wobei jede Regel $f(t_1, \ldots, t_n) \to r$ aus R in $\mathcal{T}$ mindestens einmal vorkommt.

Beispiel: Ein erweiterter definierender Baum für das parallele Oder kann wie folgt graphisch dargestellt werden, wobei der oberste eingerahmte Wurzelknoten einen Oder-Knoten darstellt:



Wie man sieht, kommt die letzte Regel der Operation "$\vee$" in dem erweiterten definierenden Baum zweimal vor.

Anzumerken ist, dass für alle Operationen eines konstruktorbasierten Termersetzungssystems immer erweiterte definierende Bäume konstruiert werden können.

Die Erweiterung von Needed Narrowing bzgl. erweiterten definierenden Bäumen ist recht einfach, wodurch wir eine lazy Narrowing-Strategie für KB-SO Termersetzungssysteme erhalten:

Definition 3.12 (Weakly Needed Narrowing) *Die Erweiterung von Needed Narrowing bzgl. erweiterten definierenden Bäumen wird* ***Weakly Needed Narrowing*** *genannt und ist wie Needed Narrowing definiert, jedoch mit folgender Erweiterung von λ für Oder-Knoten:*

$$\lambda(t, or(\mathcal{T}_1, \ldots, \mathcal{T}_k)) \supseteq \lambda(t, \mathcal{T}_i) \qquad (i = 1, \ldots, k)$$

Somit probiert die Strategie λ bei Oder-Knoten alle Alternativen aus.

Beispiel: Betrachten wir die Regeln für das parallele Oder "$\vee$" und außerdem die Regel

$$f(a) \;\rightarrow\; \texttt{True} \qquad (R_4)$$

Der auszurechnende Term sei $t = f(x) \vee f(y)$.
Dann gilt:

$$\lambda(t) = \{(1, R_4, \{x \mapsto a\}),\ (2, R_4, \{y \mapsto a\})\}$$

Daher sind

$$t \;\leadsto_{\{x \mapsto a\}}\; \texttt{True} \vee f(y)$$

und

$$t \;\leadsto_{\{y \mapsto a\}}\; f(x) \vee \texttt{True}$$

alle möglichen Weakly Needed Narrowing-Schritte für den Term t.

Weakly Needed Narrowing ist zwar nicht mehr optimal wie Needed Narrowing, aber es gilt:

Satz 3.6 ([Antoy/Echahed/Hanus 97]) *Weakly Needed Narrowing ist korrekt und vollständig für KB-SO Termersetzungssysteme bzgl. strikter Gleichheit.*

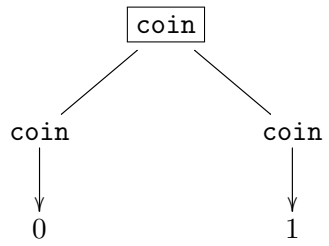
Darüber hinaus ist Weakly Needed Narrowing eine konservative Erweiterung, d.h. bei induktiv-sequentiellen Termersetzungssystemen verhält sich Weakly Needed Narrowing wie Needed Narrowing und ist damit auch optimal auf dieser Teilklasse.

3.4.3 Nichtdeterministische Operationen

Wie wir gesehen haben, ist die Voraussetzung für die Anwendung der (Weakly) Needed Narrowing-Strategie die Existenz erweiterter definierender Bäume. Dies bedeutet, dass die Eigenschaft „konstruktorbasiert" der Termersetzungssysteme wichtig ist. Wie sieht es allerdings mit der Konfluenzeigenschaft aus? Betrachten wir dazu die schon früher vorgestellte Operation `coin`, die zu 0 oder 1 ausgewertet werden kann:

```
coin = 0
coin = 1
```

Da diese trivialerweise konstruktorbasiert ist, existiert hierzu auch ein erweiterter definierender Baum, wobei der Wurzelknoten ein Oder-Knoten ist:



Weil die Auswertung von `coin` die beiden unterschiedlichen Werte 0 oder 1 liefert, spricht man auch von einer **nichtdeterministischen Operation**.
Da ein erweiterter definierender Baum für `coin` existiert, kann man offensichtlich `coin` auch mit Weakly Needed Narrowing auswerten. Was spricht also dagegen, solche nichtdeterministischen Operationen (also solche, die nicht konfluent sind) zuzulassen? In [González-Moreno *et al.* 99] wurde gezeigt, dass nichts dagegen spricht, denn man kann auch nichtdeterministischen Operationen eine präzise mathematische Bedeutung geben. Wenn z.B. eine deterministische Funktion f eine Abbildung der Art

$$f : values \to values$$

ist, denn kann eine nichtdeterministische Operation g interpretiert werden als Abbildung der Art

$$g : values \to 2^{values}$$

d.h. g bildet einen Eingabewert auf eine Menge von Ausgabewerten ab. Die genaueren Details wollen wir hier nicht diskutieren. Stattdessen wollen wir zeigen, wie solche Operationen sinnvoll zur Programmierung eingesetzt werden können (was wir schon früher einmal kurz an einem Beispiel gesehen haben).

Grundsätzlich können nichtdeterministische Operationen verwendet werden, um Berechnungen mit unterschiedlichen Ergebnissen elegant zu formulieren. Zu beachten ist auch, dass die Verwendung nichtdeterministischer Operationen keinen Mehraufwand verursacht, da eine logisch-funktionale Sprache auf Grund der logischen Anteile schon Nichtdeterminismus anbietet. Z.B. könnte die Operation `coin` auch ohne überlappende Regeln unter Verwendung einer freien Variablen definiert werden:[2]

```
coin = zeroOne x
  where
    x free

    zeroOne 0 = 0
    zeroOne 1 = 1
```

[2]Tatsächlich wurde in [Antoy/Hanus 06] gezeigt, dass überlappende Regeln und freie Variablen gleich mächtige Konzepte sind, d.h. freie Variablen können mit überlappenden Regelsystemen eliminiert werden und umgekehrt.

Beispiel: Berechnung von Permutationen

Wir haben schon in einem früheren Beispiel gesehen, wie man Permutationen einer Liste berechnen kann. Nun wollen wir eine andere Variante kennenlernen, die auf einer nichtdeterministischen Operation `insert` zum Einfügen eines Elements an einer beliebigen Stelle einer Liste beruht:

```
insert :: a → [a] → [a]
insert x ys     = x : ys
insert x (y:ys) = y : insert x ys
```

Mit dieser Operation ist die Definition einer Permutation weitgehend trivial:

```
perm :: [a] → [a]
perm []     = []
perm (x:xs) = insert x (perm xs)
```

Somit liefert `perm` *eine* beliebige Permutation der Eingabeliste. In einer Programmierumgebung, die alle Werte eines Ausdruck ausgibt, würde also z.B. `perm [1,2,3]` zu den Werten

```
[1,2,3]
[1,3,2]
[2,1,3]
[2,3,1]
[3,1,2]
[3,2,1]
```

ausgerechnet werden.

Eine mögliche Anwendung von Permutationen ist das Sortieren einer Liste durch Aufzählung aller Permutationen. Hierzu definieren wir eine Funktionen, die die Identität auf sortierten Listen ist (und auch nur für diese definiert ist):

```
sorted []              = []
sorted [x]             = [x]
sorted (x:y:ys) | x<=y = x : sorted (y:ys)
```

Damit kann die Sortierung einer Liste einfach definiert werden als Permutation, die sortiert ist:

```
sort xs = sorted (perm xs)
```

Man sollte beachten, dass diese Definition zwar kein effizienter Sortieralgorithmus ist. Es ist aber eine (ausführbare) Spezifikation des Sortierens, die man z.B. verwenden kann, um effizientere Sortieralgorithmen zu testen.

Diese Definition hat in einer nicht-strikten Sprache wie Curry einen interessanten operationalen Effekt. Wegen der lazy Auswertung wird das Argument `(perm xs)` nicht voll-

ständig ausgewertet, sondern nur bedarfsgesteuert. Hierdurch werden nicht alle Permutationen berechnet. Betrachten wir z.B. den Ausdruck

```
sort [6,5,4,3,2,1]
```

Dieser wird zunächst reduziert zu

```
sorted (perm [6,5,4,3,2,1])
```

Wegen der "needed" Strategie wird die Permutation nicht vollständig ausgerechnet, sondern nur soweit, wie es durch `sorted` verlangt wird, d.h. nur bis zu den beiden ersten Listenelementen (wobei zunächst jeweils die erste Regel für `insert` angewendet wird):

```
sorted (6 : 5 : perm [4,3,2,1])
```

Da allerdings `6<=5` nicht erfüllt ist, führt diese Berechnungsalternative zu einem Fehlschlag, so dass die Werte von "`perm [4,3,2,1]`" überhaupt nicht berechnet werden. Durch die bedarfsgesteuerte Auswertung werden hier also schon 4! Berechnungen eingespart.

Somit können wir festhalten:

> Die Kombination nichtdeterministischer Operationen und bedarfsgesteuerter Auswertung führt zu einer bedarfsgesteuerten Suche.

Diesen Effekt haben wir bei rein logischen Programmen nicht, da erst durch die Benutzung von Funktionen die bedarfsgesteuerte Auswertung sinnvoll ermöglicht wird.

Es gibt allerdings auch ein semantisches Problem bei der Anwendung von nichtdeterministischen Operationen mit nicht-strikten Strategien, denn es gibt zwei mögliche Optionen, wie nichtdeterministische Argumente einer Funktion interpretiert werden sollen:

- Die nichtdeterministische Auswahl findet zum Zeitpunkt des Funktionsaufrufs statt (dies wird auch **call-time choice** genannt).
- Die nichtdeterministische Auswahl findet zum Zeitpunkt der Auswertung des Funktionsaufrufs statt (dies wird auch **run-time choice** genannt).

Ein kleines Beispiel soll diesen Unterschied verdeutlichen. Wir betrachten die Operation

```
double x = x + x
```

und den Ausdruck "`double (1?2)`".

Call-time choice: Hier wird vor dem Aufruf von `double` der Wert des Arguments `(1?2)` (d.h. `1` oder `2`) festgelegt und dieser Wert dann innerhalb des Aufrufs verwendet. Somit sind dann `2` oder `4` die möglichen Ergebniswerte.

Run-time choice: Hier werden die möglichen Werte des Arguments `(1?2)` erst dann gewählt, wenn dieses Argument berechnet wird. Da das Argument zweimal im Rumpf

vorkommt, können dann auch zweimal unabhängig die Werte gewählt werden. Somit gibt es bei run-time choice also die folgenden mögliche Ableitungen:

```
double (1?2)  →  (1?2) + (1?2)  →  1 + (1?2)  →  1 + 1  →  2
double (1?2)  →  (1?2) + (1?2)  →  1 + (1?2)  →  1 + 2  →  3
double (1?2)  →  (1?2) + (1?2)  →  2 + (1?2)  →  2 + 1  →  3
double (1?2)  →  (1?2) + (1?2)  →  2 + (1?2)  →  2 + 2  →  4
```

Somit sind in diesem Fall `2`, `3` und `4` die möglichen Ergebniswerte.

Welche Semantik sinnvoll ist, kann man nicht absolut sagen. Run-time choice liefert zwar alle Werte bezüglich einer Auswertung mittels Termersetzung, allerdings sind dies nicht immer die intuitiv erwarteten. Zum Beispiel würde man nicht erwarten, dass bei einer Operation wie `double` als Ergebnis überhaupt ungerade Zahlen berechnet werden.

Letztendlich muss beim Entwurf einer Programmiersprache eine Entscheidung getroffen werden. Bei Curry (wie auch bei der ähnlichen Sprache TOY [López-Fraguas/Sánchez-Hernández 99]) wird die call-time choice Semantik gewählt:

- Sie erlaubt eine effiziente Implementierung mittels "sharing" (was bei nicht-strikten Sprachen für eine optimale Auswertung sowieso gemacht wird). Zum Beispiel gibt es mit "sharing" nur die beiden folgenden Ableitungen des obigen Ausdrucks:

```
double (1?2) → ·+·   →  ·+·  →  2
               ↓ ↓       ↓↓
               (1?2)      1

double (1?2) → ·+·   →  ·+·  →  4
               ↓ ↓       ↓↓
               (1?2)      2
```

- Sie passt zur lazy Auswertung mit sharing.
- Sie führt zu kleineren Suchräumen.
- Sie hat häufig die intendierte Bedeutung (vgl. das obige `double`-Beispiel).

3.4.4 Zusammenfassung

Nachdem wir nun wichtige Narrowing-Strategien vorgestellt haben (dies sind aber noch lange nicht alle: in dem Überblicksartikel [Hanus 94] werden über 20 verschiedene Narrowing-Strategien erläutert), wollen wir nun noch einen zusammenfassenden Überblick geben.

Allgemein ist Narrowing die Erweiterung von Reduktion um die Instantiierung freier Variablen, wodurch diese (unter bestimmten Einschränkungen) zu einer vollständigen Auswertungsstrategie wird, d.h. sie findet immer alle oder allgemeinere Lösungen unter der Voraussetzung, dass alle Ableitungswege untersucht werden.

Bezüglich der erlaubten Klasse von Programmen/Termersetzungssystemen und den zu lösenden initialen Ausdrücken/Constraints gibt es generell zwei Möglichkeiten:

1. Reflexive Gleichheit (wie in der Mathematik üblich) sowie terminierende, konfluente und konstruktorbasierte Termersetzungssysteme: diese kann man mittels Innermost (Basic) Narrowing und Normalisierung lösen.

2. Strikte Gleichheit (d.h. man ist an der Berechnung von Datenobjekten interessiert) und konstruktorbasierte, schwach orthogonale Termersetzungssysteme: diese kann man mittels

   - Needed Narrowing (bei induktiv-sequentiellen Termersetzungssystemen)
   - Weakly Needed Narrowing (bei nicht induktiv-sequentiellen Termersetzungssystemen)

   lösen, wobei das "sharing" von Argumenten wichtig ist.

Die wesentliche Charakteristik von Narrowing (im Gegensatz zur Reduktion) ist

- das Raten von Werten für freie Variablen
- die eventuell nichtdeterministische Auswertung von Funktionen ($\leadsto$ unendlich viele Möglichkeiten)

Dies ist allerdings nicht die einzige Möglichkeit, logisch-funktionale Programme auszuwerten. Als mögliche Alternative wurde auch vorgeschlagen, Auswertungen von Funktionen zu verzögern, statt mögliche Werte für unbekannte Argumente zu raten. Diese Strategie, genannt „Residuation", wollen wir im nachfolgenden Kapitel erläutern.

3.5 Residuation

Die Auswertungsstrategie „Residuation" basiert auf folgendem Prinzip:

> Funktionen werden nur deterministisch ausgerechnet, d.h. eine Funktionsanwendung wird nicht ausgerechnet (sondern „verzögert"), falls einige Argumente geraten werden müssen.

Falls man dennoch in einem Programm nach passenden Werten suchen will und daher Nichtdeterminismus erwünscht ist, erlaubt man *nichtdeterministische Auswertung in Prädikaten* (d.h. boolesche Funktionen), aber nicht in anderen Funktionen.

Beispiel: Betrachten wir die folgenden Regeln:

$$\begin{array}{rr} 0 + n \to 0 & \texttt{nat}(0) \to \texttt{success} \\ s(m) + n \to s(m+n) & \texttt{nat}(s(n)) \to \texttt{nat}(n) \end{array}$$

Hier ist also "+" eine Funktion und `nat` ein Prädikat. Somit werden bei Residuation Auswertungsschritte für "+" so lange verzögert, bis das erste Argument keine Variable ist.

Anfrage:	$\underbrace{x+0}$ `=:=` $s(0)$	`&` $\underbrace{\texttt{nat}(x)}$
	nicht auswerten!	auswerten, Nichtdeterminismus zulässig
	$\leadsto_{\{x \mapsto s(x_1)\}}$	$\underbrace{s(x_1)+0}$ `=:=` $s(0)$ `&` $\texttt{nat}(x_1)$
		jetzt auswerten
	$\rightarrow$	$s(x_1+0)$ `=:=` $s(0)$ `&` $\texttt{nat}(x_1)$
	$\rightarrow$	$\underbrace{x_1+0}$ `=:=` 0 `&` $\texttt{nat}(x_1)$
		nicht auswerten
	$\leadsto_{\{x_1 \mapsto 0\}}$	$\underbrace{0+0}$ `=:=` 0 `&` `success`
		auswerten
	$\rightarrow$	0 `=:=` 0
	$\rightarrow$	`success`

Hierbei wurden die folgenden Vereinfachungsregeln für `&` verwendet:

```
success & x  → x
x & success  → x
```

Diese Vereinfachungsregeln können auch als nebenläufige Auswertung beider Argumente interpretiert werden.

Die Vorteile von Residuation sind:

- Funktionen werden nur „funktional“ verwendet.
- Residuation unterstützt nebenläufige Programmierung
  (Funktionen $\approx$ Konsumenten, Prädikate $\approx$ Erzeuger):
  Synchronisation über logische Variablen (einfaches Konzept, gestützt auf formale Logik, „deklarative Nebenläufigkeit“)
- Es ist ein einfacher Anschluss externer Funktionen möglich:
  Bsp.: Anschluss einer C-Bibliothek: Behandle externe Funktionen wie normale Funktionen, verzögere den Aufruf jedoch bis alle Argumente gebunden sind

  Bsp.: Arithmetik: Statt $0/s$-Terme und explizite Funktionsdefinitionen wie in den bisherigen Narrowing-Beispielen:
  - Fasse Zahlkonstanten als Konstruktoren (unendlich viele) auf.
  - Interpretiere $x+y$ als externe Funktion, die erst aufgerufen wird, wenn x und y an Konstanten gebunden sind.

  Dies ist die Grundlage der arithmetischen Operationen, die in Curry eingebaut sind. Somit sind in Curry folgende Berechnungen möglich:

```
x=:=3 & y=:=5 & z=:=x*y   ⇝   {x=3, y=5, z=15} success

x=:=y+1 & y=:=2           ⇝   {y=2, x=3} success
```

Man beachte, dass bei der zweiten Berechnung die Auswertung des linken Gleichheitsconstraint zunächst verzögert wird.

Residuation hat allerdings auch einige Nachteile:

- Residuation ist ein unvollständiges Lösungsverfahren. Betrachten wir z.B.

  ```
  x+0 =:= s(0)
  ```

  Die Auswertung wird mit Residuation verzögert und liefert somit kein Ergebnis, wogegen mit Narrowing die Lösung $\{\texttt{x} \mapsto \texttt{s(0)}\}$ ausgerechnet wird.

  Mit Residuation wäre dieses Constraint oder auch das Constraint "`2 =:= x+1`" nur dann vollständig ausrechenbar, falls alle Funktionsargumente (wie `x`) im Berechnungsverlauf gebunden werden.

- Es kann bei Residuation durch die verzögerte Auswertung von Bedingungen sogar zu unendliche Ableitungen kommen.

  Betrachten wir als Beispiel die folgende alternative Definition der Listenumkehrung `rev` (als Prädikat):

  ```
  rev [] []     = success
  rev l  (x:xs) | rx++[x]=:=l & rev rx xs
                = success           where rx free
  ```

  Hier kommt es bei Anwendung der zweiten Klausel zu einer unendlichen Ableitung, weil die Auswertung von "`++`" verzögert wird:

  ```
  rev [0] l
  ⇝{l↦x:xs}      rx++[x]=:=[0] & rev rx xs
  ⇝{xs↦x1:xs1}   rx++[x]=:=[0] & rx1++[x1]=:=rx & rev rx1 xs1
  ⇝{xs1↦x2:xs2}  ···
  ```

  Die Ursache für dieses ungünstige Verhalten liegt in der Erzeugung neuer Bedingungen im rekursiven Fall, die alle verzögert werden ($\approx$ "passive constraints").

  Wird jedoch "`++`" mit Lazy/Needed Narrowing ausgewertet, dann ist der Suchraum endlich, da bei Bindung von `rx` an zu lange Listen die Bedingungen fehlschlagen.

  Somit ist die Verzögerung des „Ratens“ von Funktionsargumenten nicht immer besser als nichtdeterministisches Raten.

Als Zusammenfassung dieses Kapitels stellen wir noch einmal die Vor- und Nachteile von Narrowing und Residuation gegebenüber:

Narrowing:		Residuation:
+	vollständig	− unvollständig
−	Nichtdeterminismus bei Funktionen	+ Determinismus bei Funktionen
+	Optimalität bei induktiv-sequentiellen Funktionen	− ??
−	Anschluss externer Funktionen nicht möglich	+ Anschluss externer Funktionen trivial
		+ nebenläufige Programmierung

Aus diesem Grund ist es sinnvoll, beide Auswertungstechniken miteinander zu kombinieren, was die Grundlage des Berechnungsmodells der Sprache Curry ist, das wir im nächsten Kapitel vorstellen.

3.6 Ein einheitliches Berechnungsmodell für deklarative Sprachen

Wie wir gesehen haben, haben sowohl Narrowing als auch Residuation ihre Vorteile. Daher stellt sich die Frage, wie man beides sinnvoll kombinieren kann. Insbesondere muss man definieren, wann eine Berechnung verzögert wird und wo man in diesem Fall weiterrechnen soll.

Für die erste Frage kann man eine einfache Lösung angeben, indem definierende Bäume leicht erweitert werden. Hierzu fassen wir noch einmal die Kernpunkte von *Lazy Evaluation* zusammen:

- Vorteilhaft sowohl bei funktionaler und auch bei logischer Programmierung:
  - Rechnen mit unendlichen Datenstrukturen
  - Vermeidung unnötiger Berechnungen
  - normalisierende (vollständige) Strategie
- Präzise Beschreibung mit
  - case-Ausdrücken (funktionale Programmierung)
  - definierenden Bäumen (case-Ausdrücke + Oder-Knoten)

Definierende Bäume sind geeignet zur Beschreibung guter Narrowing-Strategien, aber können diese auch für Residuation genutzt werden?

Machen wir uns hierzu den Hauptunterschied zwischen Narrowing und Residuation klar:

> Falls ein verlangtes Funktionsargument eine freie Variable ist:
> Narrowing: binde Variable
> Residuation: verzögere Aufruf

Da sich verlangte Argumente immer auf branch-Knoten beziehen, kann man diesen Unterschied sehr einfach in branch-Knoten explizit machen. Hierzu erweitern wir branch-Knoten um ein Flag/Bit, das diesen Unterschied markiert. Von nun an hat ein branch-Knoten die Form

$$branch(\pi, p, r, \mathcal{T}_1, \ldots, \mathcal{T}_k)$$

wobei die Definition analog zu $branch(\pi, p, \mathcal{T}_1, \ldots, \mathcal{T}_k)$ ist und zusätzlich

$$r \in \{\texttt{rigid}, \texttt{flex}\}$$

gilt. Hierbei steht `rigid` für das Verzögern und `flex` für das sofortige Binden von Variablen.

Formal können wir nun die Strategie λ auf diesen neuen Verzweigungsknoten definieren, indem für

- $r = \texttt{flex}$ die Definition wie bisher ist (d.h. identisch zu Needed Narrowing), und für
- $r = \texttt{rigid}$ das Ergebnis der Auswertung die spezielle Konstante `suspend`, die eine verzögerte Auswertung anzeigt.

Formal erweitern wir λ (vgl. Definition 3.10) um den folgenden Fall:

$$\lambda(t, \mathcal{T}) = \texttt{suspend} \quad \text{falls } \mathcal{T} = \text{branch}(\tau, p, rigid, \mathcal{T}_1, \ldots \mathcal{T}_n) \text{ und } t|_p \text{ ist Variable}$$

Die nächste Frage, die zu beantworten ist: Wo muss man weiterrechnen, falls das Ergebnis `suspend` ist? Betrachten wir noch einmal das obige Beispiel:

```
x+0 =:= s(0)  &  nat(x)
```

Da die Auswertung des ersten Constraint suspendiert wird, muss man hier mit der Auswertung von `nat(x)` weitermachen.

Dies können wir durch folgende Erweiterung von λ auf `&`-Termen definieren:

$$\lambda(e_1 \texttt{ \& } e2) \quad = \quad \begin{cases} 1 \cdot \lambda(e_1) & \text{falls } \lambda(e_1) \neq \texttt{suspend} \\ 2 \cdot \lambda(e_2) & \text{falls } \lambda(e_1) = \texttt{suspend} \text{ und } \lambda(e_1) \neq \texttt{suspend} \\ \texttt{suspend} & \text{sonst} \end{cases}$$

wobei die Konkatenation einer Position i zu einer Narrowing-Tripelmenge S wie folgt definiert ist:

$$i \cdot S = \{(i \cdot p, r, \sigma) \mid (p, r, \sigma) \in S\}$$

Durch diese Definition wird eine Priorität für die Berechnung des linken Constraint festgelegt, aber prinzipiell kann man auch eine faire (indeterministische) Auswahl beider Constraints erlauben.

Damit können wir nun die **Auswertungsstrategie von Curry** beschreiben:

- **Auswertungsstrategie / Pattern Matching:**
  Für jede Funktion wird ein erweiterter definierender Baum erzeugt. Diese Erzeugung passiert ähnlich wie das Pattern Matching in Haskell (vgl. Kapitel 1.3), allerdings wird für induktiv-sequentielle Funktionen immer ein definierender Baum erzeugt, sodass die Auswertung mit Pattern Matching in diesem Fall optimal ist.

  Betrachten wir z.B. das Programm

  ```
  g 0 []     = 0
  g _ (x:xs) = x

  h x = h x
  ```

  und den zu berechnenden Ausdruck

  ```
  (g (h 0) [1])
  ```

  Diese Berechnung terminiert in Haskell nicht (wegen des in Kapitel 1.3 dargestellten strikten Links-Rechts-Pattern-Matching). In Curry wird dagegen das Ergebnis `1` geliefert, denn die Funktion `g` ist durch Fallunterscheidung über das zweite Argument induktiv-sequentiell.

- **Unterscheidung zwischen flexiblen und rigiden Berechnungen:**
  Die branch-Knoten haben bei den erzeugten definierenden Bäumen immer das Flag `flex`, d.h. alle benutzerdefinierten Operationen sind flexibel. Externe Funktionen, die nicht in Curry selbst definiert sind (z.B. arithmetische Operationen wie "`+`", "`*`",...), sind dagegen rigid.

  Will man als Benutzer selbst rigide Funktionen definieren (was aber nur selten notwendig ist), dann gibt es dazu zwei Möglichkeiten:

  1. Explizit verwendete `case`-Ausdrücke (und auch die damit definierte Operation `if-then-else`) sind rigid. Zum Beispiel wird mit

     ```
     rnot x = case x of True  → False
                        False → True
     ```

     eine Operation `rnot` definiert, für die der Ausdruck "`rnot x`" (wobei `x` eine freie Variablen ist) suspendiert.

  2. Expliziter Verzögerungsoperator

     ```
     ensureNotFree :: a → a
     ```

     Der Ausdruck "`ensureNotFree` e" suspendiert, so lange e zu einer ungebundenen Variablen ausgewertet wird, ansonsten wird der Wert von e (d.h. in Kopfnormalform) zurückgeliefert. Mittels dieses primitiven Operators werden externe Funktionen suspendiert.

- Der **Gleichheitstest** "==" ist wie in Haskell definiert als boolesche Funktion

  ```
  (==) :: a → a → Bool
  ```

  wie folgt vordefiniert:

  ```
  C == C = True          -- für alle 0-stelligen Konstruktoren C
  C x1 ... xn == C y1 ... yn = x1==y1 && ... && xn==yn
                         -- für alle n-stelligen Konstruktoren C
  C x1 ... xn == D y1 ... ym = False -- für alle Konstruktoren C ≠ D
  ```

  Außerdem ist "&&" wie in Haskell als sequentielle Konjunktion definiert:

  ```
  True  && x = x
  False && _ = False
  ```

  Zu beachten ist, dass "==" eine rigide Funktion ist, d.h. ein Aufruf wird suspendiert, falls ein Argument eine freie Variable ist.

- Das **Gleichheitsconstraint** "=:=" ist die Kombination der Auswertung zu Datentermen (strikte Gleichheit) und Unifikation, d.h. im Gegensatz zu "==" suspendiert ein Gleichheitsconstraint nicht, sondern wird durch Variablenbindung erfüllt, falls dies notwendig ist. Im Gegensatz zu "==" ist die genaue Bedeutung des Gleichheitsconstraint nicht als Funktion definierbar, sodass wir die folgende „Metadefinition" zur Auswertung von "e_1 `=:=` e_2" angeben:

  1. Werte e_1 und e_2 zur Kopfnormalform aus.
  2. Unterscheide nun die folgenden Fälle:
     - `x =:= y`: binde die Variable `x` an `y`, d.h. das Ergebnis ist die Substitution $\{\texttt{x} \mapsto \texttt{y}\}$ und der primitive Constraint `success`.
     - `x =:=` $C\ t_1 \ldots t_n$: Falls die Variable `x` nur in Funktionsaufrufen in $t_1, \ldots, t_n$ vorkommt (sonst Fehlschlag wegen "occur check"), binde `x` an $C\ y_1 \ldots y_n$ (wobei $y_1, \ldots, y_n$ neue Variablen sind) und löse das Constraint

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

     - $C\ t_1 \ldots t_n$ `=:= x`: Löse `x =:=` $C\ t_1 \ldots t_n$.
     - $C\ s_1 \ldots s_n$ `=:=` $C\ t_1 \ldots t_n$: Löse das Constraint

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

     - Sonst: Fehlschlag

- **Bedingte Regeln:** l `|` c `=` r
  Die generelle Strategie zur Anwendung einer bedingten Regel ist:

1. Pattern Matching der linken Seite l.
2. Beweise das Constraint c, d.h. reduziere es zu `success`.
3. Falls beide Schritte erfolgreich sind, ersetze den zu l passenden Teilausdruck durch r.

Wie schon früher erläutert, kann diese Bedeutung durch eine Transformation in eine unbedingte Regel definiert werden, d.h. die bedingte Regel

l `|` c `=` r

wird transfomiert in

l `=` `cond` c r

wobei die Operation `cond` „definiert“ ist durch:

```
cond success x = x
```

(man beachte, dass dies eigentlich keine konstruktorbasierte Definition ist).

Hierbei ist c in der Regel ein Constraint. Zur Kompatibilität mit Haskell sind auch boolesche Bedingungen erlaubt, wobei dann auch eine Folge von Bedingungen aufgeschrieben werden darf, was einem sequentiellen if-then-else entspricht. Beispielsweise ist die Definition

```
f x | x>0 = 1
    | x<0 = 0
```

äquivalent zu

```
f x = if x>0 then 1 else
       if x<0 then 0 else failed
```

- **Funktionen höherer Ordnung:**
  Dies ist analog zur rein funktionalen Programmierung (vgl. Kapitel 1.4), wobei der (nicht sichtbare) Applikationsoperator rigide ist, d.h. ein Ausdruck der Form “`x 2`” suspendiert, wenn `x` eine freie Variable ist. Als Alternative könnte man auch passende Funktionen raten (z.B. [Antoy/Tolmach 99]), was aber leicht zu großen Suchräumen führt.

Exkurs: **Modellierung nebenläufiger Objekte**
Wir wollen kurz zeigen, wie man Residuation einsetzen kann, um die Programmierung mit nebenläufige Objekte zu unterstützen. Als Beispiel betrachten wir ein Bankkonto als nebenläufiges Objekt. Dieses

- hat einen Zustand (den aktuellen Kontostand) und

- versteht Nachrichten (Einzahlen, Auszahlen, Stand)

In Curry können wir das Bankkonto als Prädikat mit einem Nachrichtenstrom und einem Kontostand als Parameter modellieren:

```
-- Type of messages for the bank account:
data Message = Deposit Int | Withdraw Int | Balance Int

account :: Int  → [Message]  → Success
account n messages = case messages of
  []                  → success     -- no more messages
  (Deposit  a : ms)   → account (n+a) ms
  (Withdraw a : ms)   → account (n-a) ms
  (Balance  b : ms)   → b=:=n & account n ms
```

Durch Verwendung von `case` suspendiert das Prädikat `account` auf dem Nachrichtenstrom, solange keine Nachricht vorhanden ist.

Wir können das Bankkontoobjekt verwenden, in dem wir ein Bankkonto mit einer freien Variable als Nachrichtenstrom erzeugen und dieses nebenläufig mit einer Operation verknüpfen, die diesen Nachrichtenstrom mit Nachrichten belegt, wie z.B.

```
account 0 ms  &  ms =:= [Deposit 100, Withdraw 20, Balance b]
```

Dies führt zu der Bindung `b=80`, was den Kontostand nach den beiden Bankkontooperationen darstellt.
Hier repräsentiert `ms` ein Server-Objekt, das man durch Nachrichten aktivieren kann. In diesem Fall wurden alle Nachrichten durch die Unifikation des Gleichheitsconstraint gesendet, aber im Prinzip kann dies auch durch eine Berechnung erfolgen, die diesen Strom schrittweise instantiiert.

Diese Technik kann auch einen Rahmen zur Erweiterung (funktional-) logischer Sprachen um nebenläufige objektorientierte Programmierung bilden, wie dies schon in Oz [Smolka 95] und prototypisch für Curry [Hanus/Huch/Niederau 01] gemacht wurde.

3.7 Erweiterungen / Anwendungen

In diesem Abschnitt wollen wir auf Erweiterungen der bisherigen vorgestellten logisch-funktionalen Programmierung vorstellen, die auch für konkrete Anwendungen nützlich sind.

3.7.1 Constraint-Programmierung

Bisher haben wir als einzige Constraints, d.h. Bedingungen zur Beschränkung des möglichen Lösungsraumes, nur das Gleichheitsconstraint "`=:=`" betrachtet. Mit diesem können wir Gleichungen über benutzerdefinierten Funktionen lösen. Zum Bespiel wird für den initialen Ausdruck

```
[1,2]++[x] =:= ys++[2,3]     where x,ys free
```

die einzige Lösung `{x=3, ys=[1]}` berechnet.

Dies funktioniert (mittels Narrowing), weil die beteiligten Funktionen durch explizite Regeln definiert sind. Allerdings funktioniert dies nicht für fest eingebaute Datentypen und ihre Operationen. Z.B. suspendiert die Auswertung des Ausdrucks

```
2+x =:= 5   where x free
```

an Stelle der Berechnung der Lösung `{x=3}`.

Notwendig hierzu wären spezielle Lösungsverfahren für arithmetische Constraints. So könnte man mit dem Gaußschen Eliminationsverfahren die Gleichung nach `x` auflösen und so eine Lösung berechnen:

```
2+x =:= 5  ⇝  x =:= 5-2  ⇝  x =:= 3
```

Dies führt zu dem Begriff der **Constraint-Programmierung**:

- erlaube in Programmen die Benutzung von Constraints für spezielle Datentypen, und
- stelle fest eingebaute Constraint-Löser (d.h. Lösungsalgorithmen) für diese Constraints bereit.

Konzeptuell ist dies leicht integrierbar in logische Sprachen, weil diese schon die Konzepte des Rechnens mit partieller Information und Lösungssuche unterstützen. In diesem Fall spricht man auch von **CLP** (Constraint Logic Programming).

Als Beispiel betrachten wir
CLP(R): Constraint-Programmierung mit arithmetischen Constraints über reellen Zahlen

- Funktionen auf Gleitkommazahlen: `+. -. *. /.`
  (der Punkt dient zu Unterscheidung von den üblichen arithmetischen Operatoren)
- Constraints: `=:= <. >. <=. >=.`
  (der Punkt dient zu Unterscheidung von den üblichen Vergleichsoperatoren)

  Die Argumente dieser Constraints sind Variablen, Gleitkommazahlen und Anwendungen der obigen Funktionen.

Normalerweise gibt es keine CLP(R)-Programmiersprache, sondern die Benutzung dieser Constraints erfolgt durch Import einer passenden Bibliothek. So stellt die Curry-Implementierung PAKCS eine Bibliothek `CLPR` bereit, die die notwendigen Definitionen enthält:

```
Prelude> :load CLPR
...
```

```
CLPR> 1.2 +. x =:= 2.5   where x free
Result: success
Bindings:
x=1.3
CLPR> y <=. 3.0 & 2.0 +.x >=. 0.0 & y -. 5.0 >=. x  where x,y free
Result: success
Bindings:
y=3.0
x=-2.0
```

Man beachte, dass bei der Lösung des ersten Ausdrucks das Gaußsche Eliminationsverfahren zur Anwendung gekommen ist, während für die Lösung des zweiten Ausdrucks, der Ungleichungen enthält, das Simplexverfahren zur Anwendung kommt. Als Constraint-Löser sind also schon recht komplexe Verfahren eingebaut.

Beispiel: **Hypothekenberechnung**
Für die Beschreibung aller nötigen Zusammenhänge beim Rechnen mit Hypotheken verwenden wir folgende Parameter:

- `p`: Kapital
- `t`: Laufzeit in Monaten
- `ir`: monatlicher Zinssatz
- `r`: monatliche Rückzahlung
- `b`: Restbetrag

Die Zusammenhänge lassen sich in Curry unter Verwendung der CLPR-Bibliothek wie folgt ausdrücken:

```
-- Import library CLPR:
import CLPR

mortgage p t ir r b | t >=. 0.0 & t <=. 1.0  -- lifetime not more than 1 month?
                    =  b =:= p *. (1.0 +. t *. ir) -. t *. r

mortgage p t ir r b | t >. 1.0               -- lifetime more than 1 month?
                    =  mortgage (p *. (1.0 +. ir) -. r) (t -. 1.0) ir r b
```

Dann können wir z.B. die notwendige monatliche Rückzahlung zur Tilgung einer Hypothek berechnen:

```
> mortgage 100000.0 180.0 0.01 r 0.0      where r free
r = 1200.168
```

Wir können aber auch die Laufzeit zur Finanzierung einer Hypothek berechnen:

```
> mortgage 100000.0 time 0.01 1400.0 0.0     where time free
time = 125.901
```

CLP(FD): Constraint-Programmierung über endlichen Bereichen

- Variablen haben einen endlichen Wertebereich (Intervall ganzer Zahlen)
- Constraints: Gleichungen, Ungleichungen, Elementbeziehungen, kombinatorische Verknüpfungen
- Anwendung: Lösung schwieriger kombinatorischer Problem (z.B. Personalplanung, Fertigungsplanung, Stundenpläne), viele industrielle Anwendungen

Da die zu lösenden Probleme typischerweise NP-vollständig sind, benutzt man im Gegensatz zu CLP(R) keinen vollständigen Lösungsalgorithmus, sondern verwendet ein unvollständiges Verfahren, wobei man mit einer Heuristik Lösungen rät:

- Man verwendet OR-Methoden (Operations Research) zur lokalen Konsistenzprüfung.
- Die globale Konsistenz wird durch Aufzählung möglicher Werte sichergestellt (Motto: "constrain-and-generate").

Allgemeines Vorgehen hierbei:

1. Definiere den möglichen Wertebereich der beteiligten Variablen.
2. Beschreibe die Constraints, die diese Variablen erfüllen müssen.
3. Zähle die Werte im Wertebereich auf, d.h. binde die Variablen an konkreten Werte. Hierduch werden Constraints aktiviert, die dann den weiteren Suchraum stark einschränken.

Beispiel: **Krypto-arithmetisches Puzzle**
Gesucht ist eine Zuordnung von Buchstaben zu Ziffern, sodass verschiedene Buchstaben verschiedenen Ziffern entsprechen und die folgende Rechnung stimmt:

```
  send
+ more
------
 money
```

In der Curry-Implementierung PAKCS steht die Bibliothek `CLPFD` zur Verfügung, mit der wir dieses Problem wie folgt lösen können:

```
-- Import library CLPFD:
import CLPFD
```

```
-- "send more money" puzzle in Curry with FD constraints:
-- The argument xs is a list of variables representing the letters.
smm xs =
   xs =:= [s,e,n,d,m,o,r,y] &   -- define the letters of the puzzle
   domain xs 0 9 &              -- define the domain of the letters
   s ># 0 &                     -- set up all constraints
   m ># 0 &
   allDifferent xs  &
                  1000 *# s +# 100 *# e +# 10 *# n +# d
   +#             1000 *# m +# 100 *# o +# 10 *# r +# e
   =# 10000 *# m +# 1000 *# o +# 100 *# n +# 10 *# e +# y &
   labeling [] xs               -- enumerate values for the letters
 where s,e,n,d,m,o,r,y free
```

Damit können wir nun eine Lösung sehr schnell berechnen:

```
> smm [s,e,n,d,m,o,r,y]  where s,e,n,d,m,o,r,y free
Result: success
Bindings:
s=9
e=5
n=6
d=7
m=1
o=0
r=8
y=2
```

Anmerkungen:

- `domain xs min max`: hierdurch wird der Wertebereich aller Variablen in der Liste xs als `[min..max]` definiert.
- Arithmetische Funktionen und Vergleichsoperatoren werden wie üblich aufgeschrieben, allerdings mit dem Suffix `#` zur Unterscheidung der Standardoperationen auf ganzen Zahlen.

  Zu beachten ist, dass `x>y` rigid ist, wohingegen `x>#y` oft schon eher überprüft wird. Z.B. suspendiert die Berechnung des Ausdrucks

  ```
  x>y && y<x
  ```

  Dagegen liefert die Berechnung des Ausdrucks

  ```
  domain [x,y] 0 1 & x>#y
  ```

  die eindeutige Lösung `{x=1, y=0}`.

- Kombinatorische Constraints haben spezielle Lösungsalgorithmen zur Einschränkung des Suchraumes, z.B. steht

  ```
  allDifferent xs
  ```

  für das Constraint „alle Variablen der Liste `xs` haben verschiedene Werte“. Prinzipiell wäre dies auch ausdrückbar durch eine Liste von „paarweise verschieden“-Constraints, aber dieses Constraint implementiert eine bessere (globalere) Einschränkung.

- `labeling s xs`: hierdurch werden nacheinander die Variablen der Liste `xs` mit Werten aus ihrem Wertebereich belegt, wodurch in der Regel Constraints aktiviert und gelöst werden. Zum Beispiel suspendiert die Berechnung des Ausdrucks

  ```
  domain [x,y] 0 1 & x/=#y
  ```

  Dagegen liefert die Berechnung des Ausdrucks

  ```
  domain [x,y] 0 1 & x/=#y & labeling [] [x,y]
  ```

  die beiden Lösungen `{x=0, y=1}` und `{x=1, y=0}`.

  Der Parameter `s` ist eine Liste verschiedener Optionen zur Auswahl von Strategien zur Belegung der Variablen. Beispielsweise wird durch die Option `FirstFail` die Variable mit dem kleinsten noch möglichen Wertebereich zuerst an einen ihrer Werte gebunden.

Als weiteres Beispiel betrachten wir das **Lösen eines Su Doku-Puzzles**:
Hierbei ist z.B. das folgende 9×9-Feld gegeben:

9			2			5		
	4			6			3	
		3						6
			9			2		
				5			8	
		7			4			3
7						1		
	5			2			4	
		1			6			9

Die Aufgabe ist, die leeren Felder mit den Ziffern 1 bis 9 so zu füllen, dass alle Zeilen, Spalte und 3×3-Felder keine doppelten Vorkommen haben.

Dies können wir in Curry lösen, in dem wir die Su Doku-Matrix als Liste von Listen (Typ `[[Int]]`) darstellen und das Constraint `allDifferent` auf die entsprechenden Teillisten anwenden:

```
import CLPFD
import List(transpose)
import Constraint(allC)

-- Solving a Su Doku puzzle represented as a matrix of numbers (possibly free
-- variables):
sudoku :: [[Int]] → Success
sudoku m =
  domain (concat m) 1 9 &             -- define domain of digits
  allC allDifferent m  &              -- all rows are different
  allC allDifferent (transpose m)  &  -- all columns are different
  allC allDifferent (squares m) &     -- all 3x3 squares are different
  labeling [FirstFail] (concat m)      -- bind digit variables
 where
  -- translate a matrix into a list of small 3x3 squares
  squares :: [[a]] → [[a]]
  squares [] = []
  squares (l1:l2:l3:ls) = group3Rows [l1,l2,l3] ++ squares ls

  group3Rows l123 = if head l123 == [] then [] else
   concatMap (take 3) l123 : group3Rows (map (drop 3) l123)
```

Hierbei ist `allC`, die Konjunktion aller Anwendungen eines Constraint auf eine Liste von Werten, wie folgt definiert:

```
allC c xs = foldr (&) success (map c xs)
```

Damit können wir ein konkretes Su Doku durch Angabe der gegebenen Felder und Variablen für die unbekannten Felder lösen:

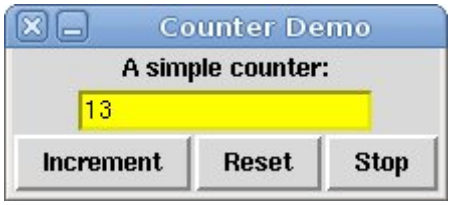
```
> sudoku [[9,x12,x13,2,x15,x16,5,x18,x19],...
x12=6
x13=8
⋮
```

Schöner ist es, ein konkretes Su Doku in einer Textmatrixdarstellung zu lesen und die Lösung formatiert auszugeben, was aber in Curry auch in vier Codezeilen erledigt werden kann.

3.7.2 GUI-Programmierung

Reale Anwendungen haben häufig graphische Benutzerschnittstellen (GUIs) zur Interaktion. Leider ist die Programmierung dieser Schnittstellen ein notwendiges, aber häufig auch aufwändiges Übel. In diesem Abschnitt wollen wir zeigen, dass die Kombination funktionaler und logischer Konstrukte geeignet ist, um die Programmierung von GUIs

einfach und abstrakt zu halten.
Betrachten wir als einfaches Beispiel einen interaktiven Zähler mit folgender GUI:



Hieraus kann man sehen, dass GUIs typischerweise wie folgt zusammengesetzt sind:

- **Widgets**: elementare Interaktionselemente wie
  - Eingabetextfeld
  - Schaltfläche (Button)
  - Textanzeige
  - Menü
  - Auswahl (check button, radio button)
  - ...
- Komposition größerer Einheiten:
  - vertikale Komposition (Spalten)
  - horizontale Komposition (Zeilen)
  - Matrixkomposition

Somit haben GUIs einen hierarchischen Aufbau, der in einer funktionalen Sprache direkt als Datenstruktur beschreibbar ist, z.B. als:

```
data Widget = Entry          [ConfItem]
            | Button         [ConfItem]
            | Label          [ConfItem]
            | TextEdit       [ConfItem]
            | Scale Int Int  [ConfItem]
            ⋮
            | Row    [ConfCollection] [Widget]
            | Col    [ConfCollection] [Widget]
            | Matrix [ConfCollection] [[Widget]]
```

Hierbei enthält der Typ `ConfItem` verschiedene Parameter für Widgets, wie z.B. Textinhalt, Hintergrundfarbe etc.:

```
data ConfItem = Text String
              | Background String
              ⋮
```

Der Typ `ConfCollection` dient zur Beschreibung der Ausrichtung bei Kompositionen:

```
data ConfCollection =
   CenterAlign | LeftAlign | RightAlign | TopAlign | BottomAlign
```

Damit könnten wir die Struktur unserer Zähler-GUI wie folgt als Datenterm spezifizieren:

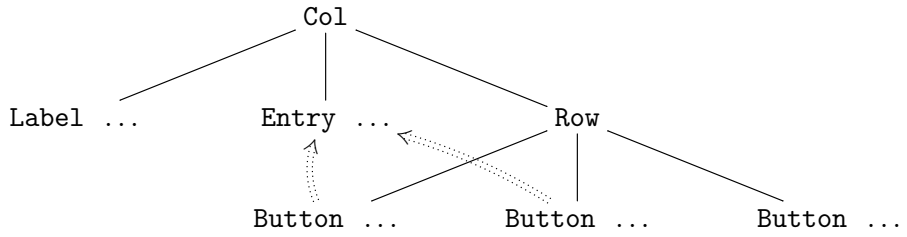
```
Col [] [
  Label [Text "A simple counter:"],
  Entry [Text "0", Background "yellow"],
  Row [] [Button [Text "Increment"],
          Button [Text "Reset"],
          Button [Text "Stop"]]]
```

Es fehlt allerdings noch die eigentliche Funktionalität der GUI. Zum Beispiel soll durch Betätigung der Increment-Schaltfläche der Text in dem `Entry`-Widget (der aktuelle Zählerstand) verändert werden. Hierzu sind zwei Dinge notwendig:

1. Die GUI-Funktionalität soll als „Event Handler“ beschrieben werden: jeder Button (oder ein anderes Widget mit Interaktionsmöglichkeiten) enthält einen „Event Handler“, d.h. eine Funktion, die bei Auslösung eines Ereignisses (durch den Benutzer) aufgerufen wird.

2. GUIs haben neben der Layout-Struktur noch eine innere logische Struktur: beispielsweise müssen die Button-Handler auf den Entry-Widget Bezug nehmen, um den Textinhalt zu ändern.

Skizze:



Wie kann man diese zusätzliche logische Struktur in einer Programmiersprache darstellen? Hierzu gibt es folgende Alternativen:

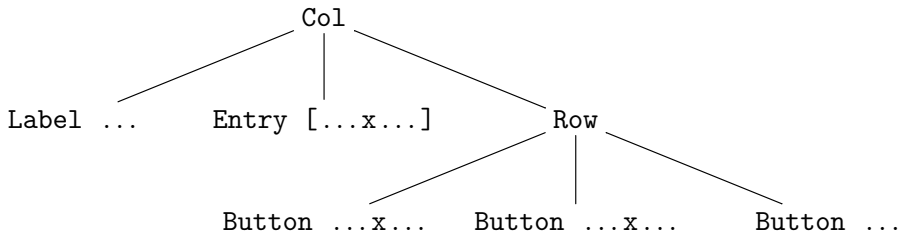
- Skriptsprachen (z.B. Tcl/Tk): Hier werden Widgets durch fest gewählte Strings identifiziert. Die Nachteile hiervon sind die Fehleranfälligkeit (z.B. Tippfehler bei Widgetnamen) und fehlende Kompositionalität (durch Namenskonflikte bei der Zusammensetzung von Widgets).

- Zeigerstrukturen (z.B. GUI-Bibliotheken in Haskell): Hier wird bei der Erzeugung eines Widgets eine Referenz auf dieses zurückgegeben. Diese Referenzen kann man dann zu größeren Einheiten zusammensetzen. Der Nachteil hiervon ist, dass eine

GUI dann schrittweise wie in einem imperativen Programm zusammengesetzt wird, statt die Struktur wie oben als hierarchische Struktur zu spezifizieren.

- In logisch-funktionalen Sprachen wie Curry gibt es eine weitere Alternative: die Verwendung logischer Variablen als Referenz.

  Skizze:




```
where x free
```

  Die Vorteile hiervon sind:

  – hierarchische Spezifikation der Layout-Struktur
  – logische Variablen, die als Referenzen benutzt werden, sind vom Compiler prüfbar (d.h. Vermeidung der Probleme der Skriptsprachen)
  – kompositionell: keine Namenskonflikte bei der Komposition von GUIs

Zur Umsetzung dieser Idee ergänzen wir den Typ der Widget-Konfigurationen:

```
data ConfItem = ...
              v...
              | WRef WidgetRef
```

Hierbei ist `WidgetRef` ein abstrakter Datentyp, der keine expliziten Konstruktoren exportiert. Aus diesem Grund kann bei der GUI-Konstruktion nur eine freie Variable als `WRef`-Parameter angegeben werden. Intern wird in der Bibliotheksimplementierung diese freie Variable mit konkreten Strings für die Kommunikation mit Tcl/Tk belegt.

Außerdem wird die Widget-Definition von `Button` so abgeändert, dass dieser als zusätzlichen Parameter einen Event Handler hat, wobei ein Event Handler eine I/O-Operation ist, die eine Verbindung zu einer GUI (Datentyp `GuiPort`) als Parameter hat:

```
data Widget = ...
            | Button (GuiPort → IO ()) → [ConfItem]
            ⋮
```

Die Manipulation einzelner Widgets in einer GUI erfolgt in der einfachsten Form durch das Lesen oder Setzen des Widget-Wertes (z.B. ist dies bei einem `Entry`-Widget der Eingabetext). Hierzu stehen die folgenden Basisoperationen zur Verfügung:

```
--- Gets the (String) value of a widget in a GUI.
getValue :: WidgetRef  → GuiPort  → IO String

--- Sets the (String) value of a widget in a GUI.
setValue :: WidgetRef  → String  → GuiPort  → IO ()

--- Updates the (String) value of a widget w.r.t. to an update function.
updateValue :: (String →String)  → WidgetRef  → GuiPort  → IO ()
updateValue upd wref gp = do
  val <- getValue wref gp
  setValue wref (upd val) gp

--- Terminating the GUI.
exitGUI :: GuiPort  → IO ()
```

Damit können wir nun unsere Zähler-GUI vollständig definieren, denn die gerade vorgestellten Elemente sind in der Curry-Bibliothek `GUI` vordefiniert:

```
import GUI
import Read(readInt)

counterGUI =
  Col [] [
    Label [Text "A simple counter:"],
    Entry [WRef val, Text "0", Background "yellow"],
    Row [] [Button (updateValue incrText val) [Text "Increment"],
            Button (setValue val "0")         [Text "Reset"],
            Button exitGUI                    [Text "Stop"]]]
 where
  val free

  incrText s = show (readInt s + 1)
```

Zur Ausführung einer GUI steht die Operation

```
runGUI :: String  → Widget  → IO ()
```

zur Verfügung, der man einen Titel und eine Widget-Spezifikation als Parameter übergibt. Somit können wir nun unsere Zähler-GUI starten:

```
> runGUI "Counter Demo" counterGUI
```

Hier hat die Verwendung einer logisch-funktionalen Sprache folgende Vorteile:

- deklarative Spezifikation von GUIs, die ausführbar ist
- algebraische Datentypen für die Layout-Beschreibung

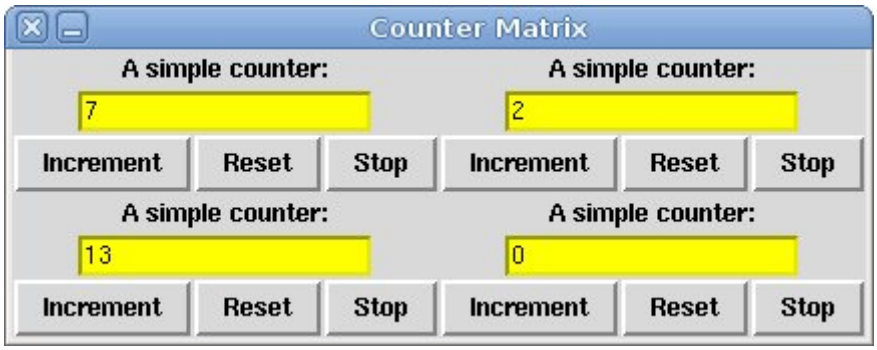
- Funktionen als Bürger 1. Klasse für Event-Handler in Datenstrukturen
- logische Variablen als prüfbare(!) Referenzen

Somit haben erst die kombinierten Konstrukte der logisch-funktionalen Programmierung die Realisierung dieser GUI-API ermöglicht.

Ein weiterer Vorteil liegt in der Kompositionalität und Wiederverwendbarkeit von GUIs. Wenn wir z.B. mehrere Zähler in einer größeren GUI benötigen, können wir einfach unsere Zähler-GUI wiederverwenden:

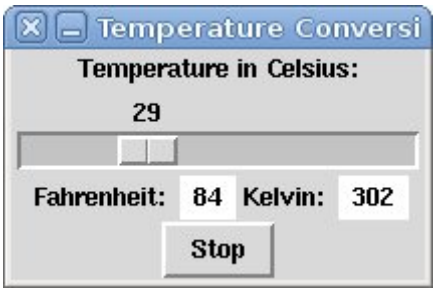
```
> runGUI "Counter Matrix"
         (Matrix [] [[counterGUI,counterGUI],[counterGUI,counterGUI]])
```

Dies führt zu einer GUI mit vier unabhängigen Zählern:



Diese Lösung wäre z.B. in Tcl/Tk problematisch wegen der festen Namen, die für jedes Widget gewählt werden müssen.

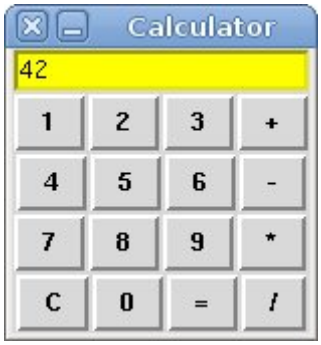
Als weiteres Beispiel betrachten wir einen Temperaturkonverter, bei dem man mit einem Schieberegler eine Temperatur in Celsius einstellen kann und diese sofort in eine Temperatur in Fahrenheit und Kelvin umgerechnet wird:



Ein Schieberegler wird durch das Widget `Scale` realisiert, das als Parameter das Minimum und Maximum der einzustellenden Werte erhält. Einem Schieberegler kann man mit dem Konfigurationsparameter `Cmd` auch einen Event-Handler zuordnen, der immer dann aktiviert wird, wenn sich an dem Wert des Schiebereglers etwas ändert. Damit können wir diese GUI wie folgt implementieren:

```
import GUI
import Read(readInt)

tempGUI =
  Col [] [
    Label [Text "Temperature in Celsius:"],
    Scale 0 100 [WRef cels, Cmd convert],
    Row [] [Label [Text "Fahrenheit: "],
            Message [WRef fahr, Background "white"],
            Label [Text "Kelvin: "],
            Message [WRef kelv, Background "white"]],
    Button exitGUI [Text "Stop"]]

 where
   cels,fahr,kelv free

   convert gp = do cs <- getValue cels gp
                   let c = readInt cs
                   setValue fahr (show (c * 9 `div` 5 + 32)) gp
                   setValue kelv (show (c + 273)) gp

main = runGUI "Temperature Conversion" tempGUI
```

Als letztes Beispiel zur GUI-Programmierung betrachten wir einen einfachen Taschenrechner mit der üblichen Funktionalität:



Das Besondere an diesem Beispiel ist die Anforderung, dass die GUI auch einen internen Zustand halten muss, der von den Event Handlern berücksichtigt wird. Zum Beispiel muss bei Drücken der Taste "=" in der Anzeige das Ergebnis erscheinen, das natürlich von den zuvor gedrückten Tasten abhängig ist.
In Curry (wie auch in Haskell) gibt es „IO-Referenzen", um veränderbare Variablen zu realisieren. Hierzu gibt es (in der Bibliothek `IOExts`) die Basisoperationen

```
--- Creates a new IORef with an initial values.
```

```
newIORef :: a  → IO (IORef a)

--- Reads the current value of an IORef.
readIORef :: IORef a  → IO a

--- Updates the value of an IORef.
writeIORef :: IORef a  → a  → IO ()
```

Zur Realisierung unserer GUI müssen wir nur noch überlegen, was die GUI in ihrem Zustand halten muss. Da wir eine Zahl eingeben und diese dann über eine Operation verknüpfen, sind zwei Dinge für den Zustand relevant:

1. Die gerade eingegebene Zahl.
2. Eine Funktion, mit der die aktuelle Zahl verknüpft werden muss, wenn man z.B. "=" drückt.

Zum Beispiel ist nach der Eingabe von "`3+4`" der aktuelle Zustand `(4,(3+))`, d.h. die aktuelle Zahl ist `4` und diese muss mit der Operation `(3+)` verknüpft werden.
Mit diesen Überlegungen wird die Implementierung der GUI recht einfach: beim Drücken einer Zifferntaste muss die aktuelle Zahl vergrößert werden, und beim Drücken einer anderen Taste muss die Verknüpfungsoperation verändert werden. In jedem Fall muss auch die Anzeige modifiziert werden. Die vollständige Implementierung der Taschenrechner-GUI ist in Abbildung 3.1 angegeben.

In ähnlicher Form kann man auch eine API mit hohem Abstraktionsniveau für die Programmierung dynamischer Web-Seiten realisieren (vgl. Curry-Bibliothek `HTML` und [Hanus 01]).

3.7.3 Eingekapselte Suche

Wenn mittels logischer Programmiertechniken mehrere Werte oder Lösungen zu einem Constraint berechnet werden, ist es oft wünschenswert, diese Werte oder Lösungen zu vergleichen oder einer weiteren Verarbeitung (z.B. Ausgabe in einer Datei) zuzuführen. Hierzu müsste man die verschiedenen Werte in eine gemeinsame Datenstruktur (z.B. eine Liste) bringen.

Mit den bisherigen Sprachmitteln ist dies nicht möglich, da die verschiedenen Werte auf unterschiedlichen Lösungswegen berechnet werden, die zunächst einmal nichts miteinander zu tun haben. Die Zusammenfassung mehrerer nichtdeterministischer Berechnungen zu gemeinsamen Strukturen nennt man auch **eingekapselte Suche**: dies bezeichnet allgemein die Möglichkeit, nichtdeterministische Berechnungen, insbesondere deren Ergebnisse, in einer Datenstruktur zu verarbeiten.

Eine eingekapselte Suche hat verschiedene Vorteile:

- Man kann verschiedene Ergebnisse abspeichern oder in einer Benutzerschnittstelle präsentieren.

```
import GUI
import Char(isDigit)
import IOExts -- use IORefs for the GUI state

-- The GUI needs a reference to the calculator state.
calcGUI :: IORef (Int,Int → Int)  → Widget
calcGUI stateref =
  Col [] [
    Entry [WRef display, Text "0", Background "yellow"],
    Row [] (map cbutton ['1','2','3','+']),
    Row [] (map cbutton ['4','5','6','-']),
    Row [] (map cbutton ['7','8','9','*']),
    Row [] (map cbutton ['C','0','=','/'])]
  where
    display free

    cbutton c = Button (buttonPressed c) [Text [c]]

    buttonPressed c gp = do
       state <- readIORef stateref
       let (d,f) = processButton c state
       writeIORef stateref (d,f)
       setValue display (show d) gp

-- Compute the new state when a button is pressed:
processButton :: Char  → (Int,Int → Int)  → (Int,Int → Int)
processButton b (d,f)
 | isDigit b = (10*d + ord b - ord '0', f)
 | b=='+' = (0,((f d) + ))
 | b=='-' = (0,((f d) - ))
 | b=='*' = (0,((f d) * ))
 | b=='/' = (0,((f d) `div` ))
 | b=='=' = (f d, id)
 | b=='C' = (0, id)

main = do
  stateref <- newIORef (0,id)
  runGUI "Calculator" (calcGUI stateref)
```

Abbildung 3.1: Die Implementierung der Taschenrechner-GUI

- Man kann verschiedene Ergebnisse vergleichen und z.B. das beste bezüglich einer Ordnung herausfiltern.
- Eingekapselte Suche ist auch notwendig bei Programmen, die Ein- und Ausgabe machen, da Ein/Ausgabe-Operationen (vgl. Kapitel 1.7) immer deterministisch sind.

In diesem Kapitel wollen wir hierzu passende Techniken vorstellen und insbesondere diskutieren, welche Probleme hierbei entstehen können.

In der logischen Programmiersprache Prolog gibt es zur eingekapselten Suche das Prädikat `findall`. Ein Aufruf der Form

```
findall(X,Goal,Results)
```

liefert in der Liste `Results` alle Werte für `X`, die bei verschiedenen erfolgreichen Berechnungen für die Anfrage `Goal` ausgerechnet wurden. Bei der Verwendung von `findall` kommt die Variable `X` also in der Anfrage `Goal` vor.

In ähnlicher Weise kann man die eingekapselte Suche auch in einer logisch-funktionalen Sprache einführen. Hierzu stellt man eine Operation

```
findall :: (a → Success) → [a]
```

zur Verfügung, die eine Constraint-Abstraktion (d.h. eine Abbildung von Werten in ein Constraint) als Argument hat und die Liste aller Werte liefert, für die dieses Constraint eine erfolgreiche Berechnung hat. Wenn wir z.B. die Vorfahrrelation `vorfahr` aus Kapitel 3.3 betrachten, dann würde durch den Aufruf

```
findall (\v → vorfahr v Peter)
```

die Liste aller Vorfahren von Peter (d.h. `[Monika,Johann,Christine,Anton]`) berechnet werden. Ebenso können wir dies auch zur Definition weiterer Operationen benutzen. Zum Beispiel könnten wir eine Operation, die die Liste aller Präfixe einer Liste berechnet, wie folgt definieren:

```
prefixesOf :: [a] → [[a]]
prefixesOf xs = findall (\p → p ++ _ =:= xs)
```

Damit würde der Ausdruck

```
prefixesOf [1,2,3]
```

zu `[[],[1],[1,2],[1,2,3]]` ausgewertet werden.

Obwohl es scheint, dass man mit `findall` eine einfache Möglichkeit zur eingekapselten Suche hat, ergeben sich bei einer genaueren Analyse einige mögliche Probleme:

1. Ein Constraint könnte unendlich viele Ergebnisse haben, wie z.B.

```
findall (\ (xs,ys)  → xs ++ [42] =:= ys)
```

(d.h. alle Listenpaare, wobei die zweite Liste im Vergleich zur ersten Liste das zusätzliche Element `42` am Ende hat).

Dies ist eigentlich kein Problem, da in nicht-strikten Sprachen Listen auch unendlich sein können, d.h. in diesem Fall könnte `findall` die unendliche Liste aller Lösungen liefern, die dann bedarfsgesteuert berechnet wird.[3]

2. Da im Prinzip nicht festgelegt ist, in welcher Reihenfolge nichtdeterministische Berechnungen abgearbeitet werden, kann `findall` eine beliebige Reihenfolge wählen. Beispielsweise könnte der obige Ausdruck

   ```
   prefixesOf [1,2,3]
   ```

   sowohl zu

   ```
   [[],[1],[1,2],[1,2,3]]
   ```

   aus auch zu

   ```
   [[1,2,3],[1,2],[1],[]]
   ```

   ausgewertet werden (oder auch zu jeder anderen Permutation dieser Liste). Tatsächlich könnte es bei einer parallelen oder nebenläufigen Auswertung aller nichtdeterministischen Verzweigungen bei wiederholten Auswertungen des gleichen Ausdrucks zu unterschiedlichen Ergebnissen kommen, was natürlich bei einer deklarativen Sprache nicht erwünscht ist.

   Dieses Problem könnte man dadurch umgehen, indem man keine Liste, sondern eine Menge oder Multimenge von Ergebnissen zurückliefert.

3. Ein weiteres Problem ist die Frage, was eigentlich eingekapselt werden soll. Betrachten wir hierzu folgendes Beispiel, bei dem alle Präfixe einer nichtdeterministischen Liste berechnet werden sollen:

   ```
   oneTwoPrefixes = prefixesOf oneOrTwo
     where oneOrTwo = [1] ? [2]
   ```

   Wenn `findall` alle nichtdeterministischen Berechnungen einkapselt, sollte es alle Präfixe der Listen `[1]` und `[2]` zurückliefern, d.h. die Liste

   ```
   [[],[1],[],[2]]
   ```

[3] Dies ist allerdings nicht bei allen Curry-Implementierungen der Fall. Z.B. übersetzt PAKCS Curry-Programme in Prolog-Programme, sodass `findall` wie in Prolog strikt ausgewertet wird.

Wenn allerdings die Liste *vor* dem Aufruf von `prefixesOf` ausgewertet wird, dann würde hierdurch eine nichtdeterministische Berechnung mit zwei Alternativen entstehen, so dass dann `prefixesOf` jeweils für jede Alternative aufgerufen wird. Betrachten wir z.B. die folgende Variante der letzten Definition:

```
oneTwoNEPrefixes | length oneOrTwo > 0 = prefixesOf oneOrTwo
  where oneOrTwo = [1] ? [2]
```

Da beide Alternativen von `oneOrTwo` nicht-leere Listen sind, sollte diese Definition gleichbedeutend mit der vorherigen sein. Durch die Auswertung von `length` entstehen allerdings zwei Berechnungszweige:

```
oneTwoNEPrefixes
→* cond (length oneOrTwo > 0) (prefixesOf oneOrTwo)
  →* cond (length ([1] ? [2]) > 0) (prefixesOf ([1] ? [2]))
    →* cond (length [1] > 0) (prefixesOf [1])
      →* prefixesOf [1] →* [[],[1]]
    →* cond (length [2] > 0) (prefixesOf [2])
      →* prefixesOf [2] →* [[],[2]]
```

Insgesamt erhalten wir hier die zwei Werte `[[],[1]]` und `[[],[2]]`, im Gegensatz zur vorherigen Definition.

Das Besondere an diesem Beispiel ist die Tatsache, dass `oneOrTwo` *außerhalb* des Aufrufes von `prefixesOf` definiert bzw. eingeführt wurde. Daher könnte man auch argumentieren, dass `findall` nur den Nichtdeterminismus einkapseln soll, der im Argument von `findall` auftaucht. Da sowohl bei `oneTwoPrefixes` als auch bei `oneTwoNEPrefixes` der Nichtdeterminismus von `oneOrTwo` außerhalb des Aufrufs eingeführt wurde, sollte dieser dann auch nicht eingekapselt werden, sodass bei beiden Varianten die zwei Ergebnisse `[[],[1]]` und `[[],[2]]` berechnet werden.

Das letzte Problem hängt mit der Kombination von eingekapselter Suche und lazy Auswertung zusammen (und taucht daher z.B. bei Prolog nicht auf). Tatsächlich wurden diese Probleme in früheren Arbeiten zur eingekapselten Suche [Hanus/Steiner 98] übersehen, so dass verschiedene Curry-Implementierungen dies unterschiedlich implementieren. Zum Beispiel liefert PAKCS bei `oneTwoPrefixes` einen Wert, wogegen MCC hier zwei alternative Werte liefert. Dieses Problem wurde erst vor einigen Jahren erkannt und wird seitdem diskutiert, im Folgenden stellen wir hierzu eine Lösung vor.

Zunächst wollen wir allerdings noch einmal die verschiedenen Ansätze zur eingekapselten Suche erläutern. Zur Vereinfachung (d.h. um nicht immer mit Constraint-Abstraktionen zu arbeiten) nehmen wir an, dass eine Operation `allValues` existiert, die die Liste aller Werte ihres Arguments liefert (wir ignorieren hier also zunächst das Problem 2 der Reihenfolge der Ergebnisse):

```
allValues :: a → [a]
```

Wir betrachten die schon früher eingeführte nichtdeterministische Operation `coin`:

```
coin = 0
coin = 1
```

Somit sollte also `allValues coin` zu der Liste `[0,1]` ausgewertet werden. In ähnlicher Weise wird der Ausdruck

```
allValues coin ++ allValues coin
```

zu der Liste `[0,1,0,1]` ausgewertet. Wenn wir allerdings den Ausdruck

```
let x = coin in allValues x ++ allValues x
```

betrachten, ist das Ergebnis nicht mehr so klar (vgl. obiges Problem 3). Hier kann man zwei Sichtweisen unterscheiden:

- Bei der **starken Einkapselung** (*strong encapsulation*) wird jeder auftretende Nichtdeterminismus eingekapselt, unabhängig davon, wo dieser herkommt. Hier wäre also das Ergebnis `[0,1,0,1]`.

  Bei der starken Einkapselung spielt also die syntaktische Form des Ausdrucks im Programm eine geringere Rolle, wichtig ist nur die Form, wenn die eingekapselte Suche gestartet wird. Allerdings kann dann, wie wir oben gesehen haben, die Auswertungsreihenfolge einen Einfluss auf das Ergebnis haben.

- Bei der **schwachen Einkapselung** (*weak encapsulation*) wird nur der Nichtdeterminismus eingekapselt, der im Argument von `allValues` auftritt, d.h. nichtdeterministische Operationen, die außerhalb definiert sind und als Parameter hineingereicht werden, werden nicht gekapselt. Hier gäbe es also die Ergebnisse `[0,0]` und `[1,1]`.

  Bei der schwachen Einkapselung ist also die syntaktische Form des Aufrufs von `allValues` relevant, allerdings hat die Auswertungsreihenfolge keinen Einfluss auf das Ergebnis. Letzteres ist bei deklarativen Sprachen wünschenswert.

Um noch einmal die Abhängigkeit der starken Einkapselung von der Auswertungsstrategie zu verdeutlichen, betrachten wir den folgenden Ausdruck:

```
let x = coin in allValues x ++ [x] ++ allValues x
```

Da der mittlere Ausdruck `[x]` zu den Werten `0` oder `1` ausgerechnet wird, könnte man mit der starken Einkapselung die beiden Gesamtergebnisse `[0,1,0,0,1]` und `[0,1,1,0,1]` erwarten. Tatsächlich wird durch die Links-Rechts-Auswertung der Konkatenationen etwas anderes ausgewertet. Nachdem das erste `allValues` zu `[0,1]` ausgewertet ist, wird nun `[x]` ausgewertet. Dadurch werden zwei alternative Berechnungen angestoßen:

- In einer Berechnung wird `x` an `0` gebunden. Da aber `x` auch im letzten Aufruf von `allValues` vorkommt, wird da also `allValues 0` ausgerechnet, was zum Ergebnis `[0]` führt. Insgesamt erhalten wir also `[0,1]++[0]++[0]`, was zu `[0,1,0,0]` ausgewertet wird.

- In einer weiteren Berechnung wird `x` an `1` gebunden, was analog zu dem Gesamtergebnis `[0,1,1,1]` führt.

Somit erhalten wir also die zwei Werte `[0,1,0,0]` und `[0,1,1,1]` an Stelle der Werte `[0,1,0,0,1]` und `[0,1,1,0,1]`. Man beachte, dass, falls der Teilausdruck `[x]` vorne oder hinten steht, wiederum andere Werte errechnet werden (welche?).

Wie man sieht, ist also das Ergebnis der starken Einkapselung sehr stark von der Auswertungsreihenfolge abhängig, so dass textuell identische Aufrufe (wie z.B. `allValues x`) im Verlaufe der Berechnung zu unterschiedlichen Ergebnissen führen. Bei der schwachen Einkapselung ist dies nicht der Fall. Hier würde das `coin`, das ja außerhalb von jedem `allValues` definiert ist, nie innerhalb von `allValues` ausgewertet, sodass wir die Ergebnisse `[0,0,0]` und `[1,1,1]` erhalten.

Ähnliche Effekte können auch bei der Verwendung logischer Variablen auftreten, denn logische Variablen sind ja sehr ähnlich zu nichtdeterministischen Operationen. Tatsächlich ist es nicht sinnvoll, logische Variablen, die außerhalb definiert werden, innerhalb einer eingekapselten Suche zu binden, denn man könnte diese ja in verschiedenen Suchzweigen an verschiedene Werte binden, so dass es außerhalb unklar wäre, welche Bindungen man betrachten soll. Dies sollte durch folgendes Beispiel klar werden:

```
boolVal False = 0
boolVal True  = 1

main | allValues (boolVal x) /= []
     = x
 where x free
```

Aus diesem Grund werden logische Variablen in einer eingekapselten Suche überhaupt nicht gebunden: wenn man den Wert einer außerhalb eingeführten logischen Variablen innerhalb einer eingekapselten Suche benötigt, wird die Suche so lange suspendiert, bis der Wert außerhalb gebunden wurde. Dieses Prinzip erinnert an Residuation, weswegen es manchmal auch als **rigide eingekapselte Suche** bezeichnet wird.

Ein Nachteil der schwachen Einkapselung ist die Abhängigkeit von der syntaktischen Struktur des jeweiligen Aufrufs, so dass scheinbar ähnliche Aufrufe zu unterschiedlichen Ergebnissen führen. Z.B. liefern die Aufrufe

```
allValues coin
```

und

```
let x = coin in allValues x
```

unterschiedliche Ergebnisse bzgl. der schwachen Einkapselung (nicht jedoch bei der starken Einkapselung). Dies ist auch der Grund, warum man nicht einfach für `allValues` weitere Abstraktionen definieren kann. Zum Beispiel könnte man auf die Idee kommen,

einen neuen Suchoperator zu definieren, der in den Ergebnissen identische Werte herausfiltert:

```
allUniqValues x = nub (allValues x)
 where
  nub []     = []
  nub (x:xs) = x : nub (filter (/=x) xs)
```

Dann liefern die Ausdrücke

```
allValues coin
```

und

```
allUniqValues coin
```

bzgl. der starken Einkapselung identische Werte, während bei der schwachen Einkapselung beim ersten Ausdruck der Wert `[0,1]` und beim zweiten Ausdruck die Werte `[0]` oder `[1]` berechnet werden (weil hier `coin` als Parameter zu `allValues` hineingereicht wird!).

Diese Abhängigkeit von der Aufrufstruktur ist zwar ein Nachteil der schwachen Einkapselung, allerdings wiegt der Vorteil der Unabhängigkeit von der Berechnungsreihenfolge doch sehr stark, sodass letztendlich die schwache Einkapselung für eine deklarative Sprache sinnvoll ist. Auf der anderen Seite ist es nützlich, wenn man klare syntaktische Konventionen hat, welche Teile bei der eingekapselten Suche wirklich eingekapselt werden und welche nicht. Macht man dies nicht deutlich, dann führt dies leicht zu Missverständnissen wie im Beispiel `allUniqValues`.

Ein Konzept, was diese Forderungen unterstützt, sind **Mengenfunktionen** oder **Set Functions** [Antoy/Hanus 09]. Nach unserer bisherigen Diskussion ist die Definition recht einfach:

Definition 3.13 (Set Function) *Für eine Funktion*

$$f \texttt{ :: } \tau_1 \to \cdots \to \tau_n \to \tau$$

ist die zugehörige Mengenfunktion oder Set Function als

$$f_{\mathcal{S}} \texttt{ :: } \tau_1 \to \cdots \to \tau_n \to 2^{\tau}$$

definiert, wobei 2^{τ} alle Mengen von Werten aus τ bezeichnet. Der Ausdruck $f_{\mathcal{S}}\ e_1 \ldots e_n$ liefert die Menge aller Werte, zu der der Ausdruck

$$\begin{array}{l} \texttt{let } x_1 \texttt{ = } e_1 \\ \quad\vdots \\ \quad x_n \texttt{ = } e_n \\ \texttt{in } f\ x_1 \ldots x_n \end{array}$$

ausgewertet wird, wobei der Nichtdeterminismus in den Argumenten $e_1, \ldots, e_n$ nicht eingekapselt wird.

Damit entspricht der Ausdruck $f_{\mathcal{S}}\ e_1 \ldots e_n$ also dem Aufruf

```
let x1 = e1
    ⋮
    xn = en
 in allValues (f x1 ... xn)
```

bezüglich der schwachen Einkapselung, wobei das Ergebnis keine Liste, sondern eine Menge ist. Der Ausdruck $f_{\mathcal{S}}\ e_1 \ldots e_n$ liefert also Wertemengen, wobei nur der Nichtdeterminismus von f, nicht jedoch von den Argumenten eingekapselt wird. Damit wird durch jede definierte Funktion eine syntaktische „Einkapselungsgrenze“ bereitgestellt. Betrachten wir hierzu folgende Definitionen:

```
coin = 0 ? 1

bigCoin = 2 ? 4

f x = coin + x
```

Dann liefert der Ausdruck "$\texttt{f}_{\mathcal{S}}$ `bigCoin`" die beiden Mengen `{2,3}` und `{4,5}` als Ergebnis.

Die Wertemengen sind ein abstrakter Datentyp, auf dem bestimmte Operationen definiert sind, wobei man aber z.B. nicht auf das „erste“ Element zugreifen kann. Die Curry-Implementierungen PAKCS und KiCS2 stellen Mengenfunktionen in Form der Bibliothek `SetFunctions` zur Verfügung. Wenn man diese importiert, erhält man die Mengenfunktion zu einer n-stelligen Funktion f durch den Ausdruck `set`n f. Die Wertemenge wird durch den polymorphen abstrakten Datentyp `Values` repräsentiert, auf dem im Modul `SetFunctions` einige Operationen zum Leerheitstest, Elementtest und weiteren Verarbeitung definiert sind, wie z.B.

```
isEmpty     :: Values a → Bool       -- leere Wertemenge?
valueOf     :: a → Values a → Bool   -- Element enthalten?
foldValues  :: (a → a → a) → a → Values a → a
mapValues   :: (a → b) → Values a → Values b
maxValue    :: (a → a → Bool) → Values a → a
minValue    :: (a → a → Bool) → Values a → a
sortValues  :: Values a → [a]
values2list :: Values a → IO [a]
printValues :: Values a → IO ()
```

Mit `sortValues` kann man eine Menge in eine sortierte Liste umwandeln. `values2list` und `printValues` wandelt die Menge ohne Sortierung um (was unproblematisch ist, weil

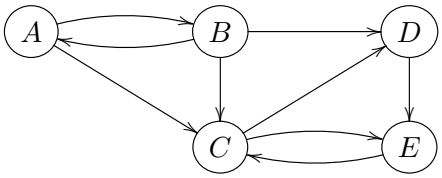
dies in der I/O-Monade stattfindet).

Somit können wir das letzte Beispiel wie folgt programmieren:

```
coin = 0 ? 1

bigCoin = 2 ? 4

f x = coin + x

main1 = sortValues (set1 f bigCoin)

main2 = foldValues (+) 0 (set1 f bigCoin)
```

Hier wird `main1` zu den beiden Listen `[2,3]` und `[4,5]` ausgewertet, und `main2` wird zu den Werten `5` und `9` ausgewertet.

Eine Motivation zur eingekapselten Suche war der Wunsch, innerhalb eines Programms verschiedene Lösungen zu vergleichen, um z.B. die beste auszuwählen. Wir wollen nun an einem Beispiel zeigen, wie wir dies mit Mengenfunktionen realisieren können. Als Beispiel betrachten wir das **Suchen eines kürzesten Weges** in einem Graphen. Unser Beispielgraph hat die folgende Struktur:



Um diese Struktur in einem Programm zu modellieren, definieren wir zunächst einen Datentyp zur Darstellung der Knoten:

```
data Node = A | B | C | D | E
```

Zur Darstellung der Kanten gibt es verschiedene Möglichkeiten. In einer logisch-funktionalen Sprache bietet sich die Darstellung als nichtdeterministische Operation an, d.h. wir definieren eine Operation `succ`, die einen Nachfolger zu einem Knoten liefert:

```
succ :: Node → Node
succ A = B
succ A = C
succ B = A
succ B = C
succ B = D
succ C = D
succ C = E
succ D = E
succ E = C
```

Nun definieren wir eine Operation, die uns (nichtdeterministisch) einen Pfad zwischen zwei Knoten liefert. Dies kann z.B. dadurch erfolgen, dass wir einen Pfad vom Ausgangsknoten schrittweise erweitern, bis wir den Zielknoten erreicht haben:

```
pathFromTo :: Node  → Node  → [Node]
pathFromTo n1 n2 = extendPath n2 [n1]

extendPath :: Node  → [Node]  → [Node]
extendPath target (n:p) =
  if target==n then reverse (n:p)
               else extendPath target ((succ n) : (n:p))
```

Dieses Programm hat allerdings das Problem, dass es bei zyklischen Graphen (so wie in unserem Beispiel) in eine Endlosschleife gerät. Wir können dies vermeiden, in dem wir nur zyklenfreie Pfade erzeugen. Hierzu gibt es eine Programmiertechnik, die auch als „Constrained Constructor“ bezeichnet wird. Wir tauschen den Listenkonstruktor “`:`” einfach gegen einen Konstruktor `consNoCycle` aus, der nur zyklenfreie Pfade konstruiert und ansonsten fehlschlägt:

```
extendPath :: Node  → [Node]  → [Node]
extendPath target (n:p) =
  if target==n then reverse (n:p)
               else extendPath target (consNoCycle (succ n) (n:p))

consNoCycle :: Node  → [Node]  → [Node]
consNoCycle n p | all (n /=) p
                = n:p
```

Mittels Mengenfunktionen können wir nun einfach einen kürzesten Pfad konstruieren:

```
shortestPath :: Node  → Node  → [Node]
shortestPath n1 n2 = minValue shorter ((set2 pathFromTo) n1 n2)
 where
   shorter p1 p2 = length p1 <= length p2
```

Damit liefert der Aufruf “`shortestPath A E`” den Wert `[A,C,E]`.

Als weiteres Beispiel betrachten wir das **8-Damen-Problem**. Es sollen 8 Damen auf einem Schachbrett so angeordnet werden, dass sie sich gegenseitig nicht schlagen können. Da eine Dame horizontal und vertikal schlagen kann, ist es klar, dass alle horizontalen und alle vertikalen Positionen der Damen verschieden sein müssen. Somit können wir annehmen, dass die n. Dame in Zeile n steht und uns auf die Frage konzentrieren, in welcher Spalte die n. Dame steht. Da alle Spalten der Damen verschieden sein müssen, müssen also die Spaltenpositionen eine Permutation der Reihenpositionen sein. Die Definition einer Permutation übernehmen wir von früher:

```
perm [] = []
perm (x:xs) = insert x (perm xs)
 where
  insert z ys     = z : ys
  insert z (y:ys) = y : insert z ys
```

Eine Permutation ist allerdings nur eine Lösung, wenn jede Dame eine andere nicht diagonal schlagen kann. Dies können wir einfach dadurch ausdrücken, indem wir definieren, was eine „unsichere“ Positionspermutation ist: dies ist eine, bei der die Spaltenpositionen den gleichen Abstand wie die Zeilenpositionen haben:

```
unsafe xs | xs =:= _++[x]++y++[z]++_
          = abs (x-z) =:= length y + 1
  where x,y,z free
```

Hierbei bestimmt also die Länge des „Zwischenraums“ `y` den Abstand der Damen. Damit ist also eine Permutation eine Lösung, wenn sich keine Damen schlagen können, d.h. wenn es keine Lösung für `unsafe` gibt. Diese Negation (in der logischen Programmierung auch als “negation as failure” bekannt) kann mittels Mengenfunktionen und einem Leerheitstest formuliert werden:

```
queens n | isEmpty ((set1 unsafe) p) = p
 where p = perm [1..n]
```

Man beachte hierbei, dass die Definition der Permutation mittels `where` wichtig ist, denn es soll ja die ausgegebene Permutation genau die sein, die auch als sicher getestet wurde. Ebenso ist die schwache Einkapselung hier unbedingt notwendig, denn die verschiedenen Permutationen sollen gerade nicht in `unsafe` eingekapselt werden. Damit erhalten wir z.B. die folgenden Ergebnisse:

```
> queens 4
[3,1,4,2]
[2,4,1,3]
> queens 8
[8,4,1,3,6,2,7,5]
[8,3,1,6,2,5,7,4]
...
```

3.7.4 Funktionale Muster

Die Konzepte logisch-funktionaler Programmiersprachen ermöglichen eine interessante Erweiterung des üblichen Pattern Matching, die zu mächtigen Programmregeln führt. Dies wollen wir in diesem Kapitel erläutern.

Betrachten wir zunächst noch einmal die logisch-funktionale Definition des letzten Elementes einer Liste:

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

Hierbei steht "`=:=`" für *strikte* Gleichheit, d.h. die Bedingung ist erfüllt, wenn beide Seiten zu identischen Datentermen ausgewertet werden können. Dies bedeutet, dass das Argument `xs` *vollständig* ausgewertet wird, obwohl wir ja eigentlich nur das letzte Element der Liste berechnen wollen. Dies kann nicht nur zu Effizienzproblemen führen, sondern es kann auch dazu führen, dass das letzte Element nicht berechnet werden kann, falls die Berechnung anderer Listenelemente fehlschlägt. Zum Beispiel kann mit dieser Definition der Ausdruck

```
last [failed,2]
```

nicht zu `2` ausgewertet werden.

Intuitiv wertet diese Version von `last` das Argument weiter aus als eigentlich notwendig ist. Wir könnten aber `last` auch als rein funktionales Programm definieren:

```
last :: [a]  → a
last [x] = x
last (x:y:ys) = last (y:ys)
```

Dieses Programm hat die obigen Probleme nicht und ist auch recht kurz, aber die Tatsache, dass hierdurch wirklich immer das letzte Element einer Liste berechnet wird, ist nicht so offensichtlich. Man müsste hierzu noch beweisen, dass für alle Listen `xs` und Werte `x` die Eigenschaft

```
last (xs++[x]) →* x
```

gilt. Anstatt nun diese Eigenschaft zu beweisen, kann man mittels *funktionaler Muster* dies als Programmregel zur Definition von `last` benutzen:

```
last :: [a]  → a
last (xs++[x]) = x
```

Man beachte, dass in allgemeinen Termersetzungssystemen eine solche Regel erlaubt ist, allerdings nicht in üblichen funktionalen Programmen, da das Argument der linken Seite eine definierte Funktion enthält (die Definition ist also nicht konstruktorbasiert). Ein **funktionales Muster** (**functional pattern**) ist somit ein Muster, das neben Variablen und Konstruktoren auch definierte Funktionen enthält.[4]

Funktionale Muster führen nicht nur zu gut lesbaren Programmen (es sind eher „ausführbare Spezifikationen“) sondern auch zu einem besseren Programmverhalten (sie sind weniger strikt), wie wir noch sehen werden. Was bedeuten aber diese Muster und wie

[4]Funktionale Muster werden in den Curry-Implementierungen PAKCS und KiCS2 unterstützt.

werden diese abgearbeitet?

Zur Vermeidung der Definition einer neuen Programmlogik für logisch-funktionale Programme mit funktionalen Muster wird in [Antoy/Hanus 05] vorgeschlagen, solche Programme in Standardprogramme ohne funktionale Muster zu transformieren. Hierzu wird ein funktionales Muster interpretiert als Abkürzung für die Menge aller Konstruktorterme, die aus diesem Muster ableitbar sind (wobei „ableitbar“ Auswertung für logisch-funktionale Programme bedeutet). Zum Beispiel kann man mittels Narrowing das Muster `xs++[x]` wie folgt ableiten:

```
xs++[x]  ⇝{xs↦[]}       [x]
xs++[x]  ⇝{xs↦[x1]}     [x1,x]
xs++[x]  ⇝{xs↦[x1,x2]}  [x1,x2,x]
...
```

Somit steht das Muster `xs++[x]` also für die unendlich vielen Muster `[x]`, `[x1,x]`, `[x1,x2,x]`,... und die Regel

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

steht damit für die unendlich vielen Regeln

```
last [x] = x
last [x1,x] = x
last [x1,x2,x] = x
...
```

Wir können zwar die Transformation nicht explizit, d.h. zur Übersetzungszeit durchführen, da wir ein unendlich großes Programm erhalten würden, allerdings kann diese Transformation auch zur Laufzeit des Programms realisiert werden, indem die Muster dann transformiert werden, wenn es notwendig ist.

Ein Vorteil dieser Transformation ist nun offensichtlich: Das Argument von `last` muss nun nicht mehr vollständig ausgewertet werden, sondern `xs` und `x` repräsentieren Mustervariablen, die auch an unausgewertete Ausdrücke gebunden werden können (im Gegensatz zu logischen Variablen!). So kann nun der Ausdruck

```
last [failed,2]
```

mittels der zweiten transformierten Regel zu `2` ausgewertet werden.

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 3.14 (Ebenenabbildung) *Eine **Ebenenabbildung** (**level mapping**) m für ein Programm P ist eine Abbildung von Funktionsnamen, die in P definiert sind, zu natürlichen Zahlen, so dass 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 $m(\texttt{++}) = 0$ and $m(\texttt{last}) = 1$ 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 3.15 (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 ganz analog zur Standardunifikation "`=:=`" abgearbeitet (vgl. Kapitel 3.6), allerdings wird zunächst nur das linke Argument zur Kopfnormalform ausgewertet. Falls dieses eine Variable ist, wird sie 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 Berechnungsfolge:

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

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 :: [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]  → Success
someDescending (_++x:y:_) | x > y = success
```

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 3.7.3), 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 :: [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 von funktionalen Mustern. 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.

XML-Programmierung mit funktionalen Mustern

XML ist heutzutage ein wichtiges Format zum Austausch von Dokumenten. XML ist eine sogenannte **Auszeichnungssprache** (**markup language**), mit der man in Dokumenten bestimmte Strukturen deutlich machen kann. Die Auszeichnungselemente haben einen Namen (tag), der in spitzen Klammern eingefasst wird. Beispielsweise könnte das folgende XML-Dokument ein Auszug aus einer Kontaktliste sein:

```
<contacts>
  <entry>
    <name>Hanus</name>
    <first>Michael</first>
    <phone>+49-431-8807271</phone>
    <email>mh@informatik.uni-kiel.de</email>
    <email>hanus@email.uni-kiel.de</email>
  </entry>
  <entry>
    <name>Smith</name>
    <first>William</first>
    <nickname>Bill</nickname>
    <phone>+1-987-742-9388</phone>
  </entry>
</contacts>
```

Der Vorteil dieses Formats liegt in dem standardisierten Austausch und der einheitlichen Verarbeitung der Dokumente. Zum Beispiel kann ein Datenbanksystem die Ergebnisse einer Anfrage im XML-Format exportieren, sodass andere Programme diese einfach einlesen und die Strukturen weiter verarbeiten können. Beispielsweise liefert das an der CAU Kiel eingesetzte UnivIS seine Daten auch im XML-Format aus.

In der Praxis können dabei allerdings Schwierigkeiten auftreten, insbesondere wenn die konkreten Datenformate eine komplizierte Strukturen besitzen oder sich die Struktur über die Zeit ändert:

- Obwohl für viele Bereiche konkrete XML-Sprachen definiert sind, sind diese oft sehr komplex, sodass es aufwändig ist, diese zu verarbeiten, wenn man nur an einigen Teildaten interessiert ist.
- Manchmal gibt es auch keine präzise XML-Spezifikation oder diese verändert sich mit der Einsatzzeit der Systeme, sodass man die zugehörigen Programme zur XML-Verarbeitung anpassen muss.

Bei Vorliegen des letzten Punktes spricht man daher auch von „semi-strukturierten“ Daten. Dies ist z.B. bei dem obigen Beispieldokument der Fall: der erste Eintrag hat zwei E-Mail-Adressen und keinen Spitznamen, während der zweite Eintrag keine E-Mail-Adresse, dafür aber einen Spitznamen hat. Bei der präzisen Verarbeitung dieser Daten, z.B. mittels Pattern Matching, müsste man alle diese Fälle abdecken.

Um dem Programmierer die Arbeit zu erleichtern, gibt es verschiedene Sprachen und Werkzeuge zur Verarbeitung von XML-Dokumenten, wie z.B.

- XPath[5]: Pfadausdrücke zur Adressierung von Teildokumenten (wie navigiere ich von der Wurzel zum gewünschten Teilausdruck?)
- XQuery[6]: Selektion von XML-Dokumenten aufbauend auf XPath
- XSLT[7]: Transformation von XML-Dokumenten aufbauend auf XPath

All dies sind mächtige Werkzeuge, aber wegen der Verwendung von XPath eher imperativ (*wie* gehe ich...?) und für einfache Anwendungen manchmal zu aufwändig zu benutzen.

Als weitere Alternative wurde die Sprache Xcerpt [Bry/Schaffert 02] vorgeschlagen, die auf Ideen der Logikprogrammierung basiert und Pattern Matching auf partiellen Datentermen (d.h. Termen, deren Struktur nicht vollständig spezifiziert wird) erlaubt. Nachfolgend wollen wir zeigen, wie man eine ähnliche Idee mit geringem Aufwand mittels funktionaler Muster umsetzen kann.

XML-Dokumente sind nichts anderes als hierarchische Datenstrukturen, d.h. Bäume bzw. Termstrukturen. Daher kann man diese einfach durch den folgenden Datentyp repräsentieren:

```
data XmlExp = XText String
            | XElem String [(String,String)] [XmlExp]
```

Eine XML-Dokument oder XML-Ausdruck (`XmlExp`) ist somit entweder ein Text oder eine Struktur mit einem Namen, Attributen (Liste von Stringpaaren) und einer Liste von darin enthaltenen XML-Dokumenten (andere spezielle XML-Elemente werden hier der Einfachheit halber ignoriert).

Um XML-Dokumente in einem Programm einfacher aufzuschreiben, definieren wir folgende Abstraktionen:

```
xtxt :: String  → XmlExp
xtxt s = XText s

xml :: String  → [XmlExp]  → XmlExp
xml t xs = XElem t [] xs
```

Hiermit können wir nun die zweite `entry`-Struktur aus dem obigen Beispiel als folgenden Term definieren:

```
xml "entry" [xml "name"     [xtxt "Smith"],
             xml "first"    [xtxt "William"],
```

[5] `http://www.w3.org/TR/xpath`
[6] `http://www.w3.org/XML/Query/`
[7] `http://www.w3.org/TR/xslt`

```
            xml "nickname" [xtxt "Bill"],
            xml "phone"    [xtxt "+1-987-742-9388"]]
```

Das Curry-Modul `XML` enthält diese Definitionen zusammen mit weiteren nützlichen Operationen, z.B. zum Lesen von XML-Textdokumenten:

```
parseXmlString :: String  → [XmlExp]
readXmlFile    :: String  → IO XmlExp
```

Im Prinzip sind diese Elemente ausreichend, um XML-Dokumente zu verarbeiten, d.h. um Teilinformation zu extrahieren oder zu transformieren. Zum Beispiel können wir den Namen und die Telefonnummer aus einer `entry`-Struktur mittels Pattern Matching wie folgt extrahieren (man beachte, dass wir hier für eine besser lesbare Regel funktionale Muster verwenden):

```
getNamePhone
  (xml "entry" [xml "name"  [xtxt name],
                _,
                xml "phone" [xtxt phone]])  =  name ++ ": " ++ phone
```

Diese Lösung hat allerdings einige Nachteile:

- Die exakte Struktur der XML-Dokumente muss vollständig bekannt sein. So kann `getNamePhone` nur auf Einträge mit genau drei Komponenten angewendet werden, sodass sie auf beide obigen `entry`-Strukturen nicht erfolgreich anwendbar ist.
- In großen XML-Dokumenten sind viele Teile für die jeweilige Anwendung irrelevant. Trotzdem muss man Muster für das komplette Dokument angeben.
- Wenn sich die Struktur des Gesamtdokuments ändert, muss man viele Muster entsprechend anpassen, was leicht zu Fehlern führen kann.

Diese Probleme können mittels mächtigerer funktionaler Muster vermieden werden. Nachfolgend geben wir einige Muster hierzu an.

Partielle Muster

Mittels funktionaler Muster können wir z.B. an Stelle der exakten Liste aller Teildokumente nur die relevanten angeben und die übrigen weglassen:

```
getNamePhone
  (xml "entry"
    (with [xml "name"  [xtxt name],
           xml "phone" [xtxt phone]]))  =  name ++ ": " ++ phone
```

Die Bedeutung des Operators "`with`" ist, dass die angegebenen Teildokumente vorkommen müssen, aber dazwischen beliebig viele andere Elemente stehen dürfen. Diesen können wir in Curry wie folgt definieren:

```
with :: [a]  → [a]
with []     = _
with (x:xs) = _ ++ x : with xs
```

Somit wird z.B. der Ausdruck "`with [1,2]`" zu jeder Liste der Form

$$x_1 : \ldots : x_m : 1 : y_1 : \ldots : y_n : 2 : zs$$

ausgewertet, wobei x_i, y_j, zs neue logische Variablen sind. Somit passt die Definition von `getNamePhone` auf jede `entry`-Struktur, die die `name`- und `phone`-Strukturen als Kinder enthält. Solch ein **partielles Muster** ist also robust gegen weitere Teilstrukturen, die vielleicht in zukünftigen XML-Dokumenten hinzugefügt werden.

Ein Nachteil dieser Definition von `getNamePhone` ist, dass diese nur auf XML-Strukturen mit leerer Attributliste passt. Falls wir an XML-Strukturen mit beliebigen Attributen interessiert sind, könnten wir die Operation

```
xml' :: String  → [XmlExp]  → XmlExp
xml' t xs = XElem t _ xs
```

definieren, die ein XML-Dokument mit beliebigen Attributen beschreibt. Damit könnten wir z.B. die Operation `getName` definieren:

```
getName (xml' "entry" (with [xml' "name" [xtxt n]])) = n
```

Diese liefert den Namen in einer `entry`-Struktur unabhängig davon, ob die Strukturen Attribute enthalten.

Ungeordnete Muster

Wenn die Struktur der Daten bei der Entwicklung eines Softwaresystems verändert wird, könnte sich auch die Reihenfolge der Elemente in einem XML-Dokument ändern. Um unsere Programme dagegen robust zu machen, können wir festlegen, dass die Teildokumente in einer beliebigen Reihenfolge auftreten dürfen, indem wir den Operator "`anyorder`" davor setzen:

```
getNamePhone
  (xml "entry" (with
    (anyorder [xml "phone" [xtxt phone],
               xml "name"  [xtxt name]])))  =  name ++ ": " ++ phone
```

Hierbei berechnet `anyorder` eine beliebige Permutation der Argumentliste, sodass wir den Code hierfür schon kennen:

```
anyorder :: [a]  → [a]
anyorder []     = []
anyorder (x:xs) = insert (anyorder xs)
 where
```

```
insert z ys     = z : ys
insert z (y:ys) = y : insert z ys
```

Die letzte Definition von `getNamePhone` passt also auf beide `entry`-Strukturen unseres Beispieldokumentes.

Tiefe Muster

Wenn man Informationen in einem tief verschachtelten XML-Dokument sucht, dann ist es unschön, den kompletten Pfad von der Wurzel bis zum Teildokument anzugeben. Besser wäre es, nur die Teildokumente zu spezifizieren und ein Matching in beliebiger Tiefe zuzulassen. Hier nutzen wir die Operation "`deepXml`", die an Stelle von `xml` in einem Muster verwendet werden kann und passt, falls dies an einer beliebigen Stelle im gegebenen Dokument vorkommt. Wenn wir z.B. `getNamePhone` durch

```
getNamePhone
  (deepXml "entry"
     (with [xml "name"  [xtxt name],
            xml "phone" [xtxt phone]]))  =  name ++ ": " ++ phone
```

definieren und auf das komplette Beispieldokument anwenden, dann erhalten wir zwei berechnete Werte. Die Menge oder Liste aller berechneten Werte könnten wir dann mittels eingekapselter Suche erhalten.

Die Implementierung von `deepXml` ist ähnlich wie die von `with`, allerdings berechnen wir alle Strukturen, die das angegebene Element als Wurzel oder Nachfolger enthalten:

```
deepXml :: String  →  [XmlExp]  →  XmlExp
deepXml tag elems = xml' tag elems
deepXml tag elems = xml' _ (_ ++ [deepXml tag elems] ++ _)
```

Negierte Muster

Manchmal ist es von der Anwendung her sinnvoll, XML-Dokumente zu suchen, die bestimmte Muster nicht erfüllen. Zum Beispiel könnte man alle Einträge heraussuchen, die keine E-Mail-Adresse haben. Da die allgemeine Negation von Berechnungen in der logisch-funktionalen Programmierung schwierig ist, könnten wir stattdessen konstruktiv vorgehen und eine Operation "`withOthers`" definieren, die sich wie "`with`" verhält, allerdings in einem zweiten Argument die Teildokumente liefert, die *nicht* im Muster enthalten sind. Auf diesen kann man dann weitere Bedingungen formulieren. Wenn wir z.B. den Namen und Telefonnummer eines `entry`-Dokumentes wissen wollen, das keine E-Mail-Adresse hat, dann können wir dies wie folgt definieren:

```
getNamePhoneWithoutEmail
  (deepXml "entry"
    (withOthers [xml "name" [xtxt name], xml "phone" [xtxt phone]] others))
```

```
  | "email" `noTagOf` others  =  name ++ ": " ++ phone
```

Hierbei ist das Prädikat `noTagOf` erfüllt, falls das angegebene Element nicht vorkommt (die Operation `tagOf` liefert die Markierung eines XML-Dokumentes):

```
noTagOf :: String  → [XmlExp]  → Bool
noTagOf tag = all ((/=tag) . tagOf)
```

Somit liefert `getNamePhoneWithoutEmail` für unser Gesamtdokument genau einen Wert zurück.

Die Implementierung von `withOthers` ist etwas komplizierter als `with`, da wir die übrigen Element akkumulieren müssen:

```
withOthers :: [a]  → [a]  → [a]
withOthers ys zs = withAcc [] ys zs
 where -- Accumulate remaining elements:
   withAcc prevs [] others | others =:= prevs ++ suffix = suffix
                                                 where suffix free
   withAcc prevs (x:xs) others =
      prefix ++ x : withAcc (prevs ++ prefix) xs others
                                                 where prefix free
```

Somit wird der Ausdruck "`withOthers [1,2]` *os*" zu jeder Liste der Form

$x_1{:}\ldots{:}x_m{:}1{:}y_1{:}\ldots{:}y_n{:}2{:}zs$

ausgerechnet, wobei $os = x_1{:}\ldots{:}x_m{:}y_1{:}\ldots{:}y_n{:}zs$. Wenn wir diesen Ausdruck also als funktionales Muster verwenden, dann passt dieser auf jede Liste, die die Elemente `1` und `2` enthält, wobei die Variable *os* an die Liste der übrigen Dokumente gebunden wird.

Transformation von XML-Dokumenten

Da Transformationaufgaben eine der Stärken deklarativer Programmierung ist, ist natürlich auch die Transformation von XML-Dokumenten einfach nach dem Schema

transform pattern = *newdoc*

möglich. Zum Beispiel könnten wir ein `entry`-Dokument in eine andere XML-Struktur übersetzen, die die Telefonnummer und den vollen Namen der Person enthält:

```
transPhone (deepXml "entry" (with [xml "name"  [xtxt n],
                                   xml "first" [xtxt f],
                                   xml "phone" phone])) =
   xml "phonename" [xml "phone" phone, xml "fullname" [xtxt (f++' ':n)]]
```

Da die Anwendung von `transPhone` auf unser Beispieldokument nichtdeterministisch zwei neue XML-Dokumente liefert, möchte man diese zu einem neuen Dokument zusam-

menfassen. Mittels eingekapselter Suche ist dies einfach möglich. Beispielsweise liefert der Ausdruck

```
xml "table" (sortValues ((set1 transPhone) c))
```

eine vollständige Tabelle aller Paare von Telefonnummern und vollen Namen aus dem Dokument c.

Ebenso können wir auch Mengenfunktionen schachteln, um Zwischeninformationen zu akkumulieren. Als Beispiel könnten wir eine Liste aller Personen mit der Anzahl ihrer E-Mail-Adressen erstellen. Hierzu definieren wir eine Operation, die für ein `entry`-Dokument den Namen und die Anzahl der E-Mail-Adressen liefert:

```
getEmails (deepXml "entry" (withOthers [xml "name" [xtxt name]] os))
          = (name, length (sortValues ((set1 emailOf) os)))
 where
   emailOf (with [xml "email" email]) = email
```

Die vollständige Liste dieser Einträge können wir aus dem Dokument c wie folgt berechnen:

```
sortValues ((set1 getEmails) c)
```

Für unser Beispieldokument wird damit `[("Hanus",2),("Smith",0)]` ausgerechnet.

Anwendung

Diese Art der XML-Verarbeitung zeigt noch einmal die generellen Vorteile der deklarativen Programmierung. Durch die Kombination mächtiger Konzepte wie musterorientierte Programmierung, Nichtdeterminismus und Suche erhält man einfache, lesbare und ausführbare Problemlösungen. Da die Lesbarkeit und Effizienz der Programmentwicklung im Vordergrund stehen, aber zunächst einmal nicht die Effizienz der Ausführung, hängt es immer von der konkreten Anwendung ab, ob die Laufzeiteffizienz ausreichend ist. Obwohl man nicht erwarten kann, dass man in den Laufzeiten mit spezialisierten XML-Implementierungen konkurrieren kann, kann dies trotzdem für viele Anwendungen ausreichend sein. Zum Beispiel ist es bei unserer Implementierung so, dass häufig die Zeit zum (deterministischen!) Einlesen eines XML-Dokumentes länger ist als die Zeit zu dessen (nichtdeterministischer) Verarbeitung. So wird diese einfache Implementierung auch eingesetzt, aus dem UnivIS der CAU Kiel[8] die Veranstaltungsdaten im XML-Format zu extrahieren und in die Moduldatenbank des Instituts[9] automatisch einzupflegen. In dieser Anwendung ist es sehr nützlich, tiefe und partielle XML-Muster zu spezifizieren, um die große Datenmenge der UnivIS-XML-Dokumente, von denen der Großteil ignoriert wird, einfach zu verarbeiten.

[8] `http://univis.uni-kiel.de/`
[9] `https://mdb.ps.informatik.uni-kiel.de/`

4 Zusammenfassung

Deklarative Sprachen basieren auf folgenden Prinzipien:

- definiere Eigenschaften der Objekte
- abstrahiere von Ausführungsdetails (z.B. Speicherverwaltung, aber auch von Ausführungsreihenfolgen)
- mehr Freiheitsgrade für die Ausführung und Optimierung
  (aber: eine konkrete Programmiersprache legt eine Strategie fest!)
- „gute“ Auswertungsstrategien sind wichtig

In dieser Vorlesung haben wir insbesondere die folgenden Aspekte behandelt:

- Funktionale Sprachen: Reduktionsstrategien
- (Funktionale) Logische Sprachen: Reduktionsstrategien + Variablenbindung
- Aspekte für flexible, wiederverwendbare SW-Entwicklung:
  - Funktionen höherer Ordnung
  - flexible Typsysteme
- Aspekte für zuverlässige und wartbare Software:
  - lokale Definition (Gleichungen, Pattern Matching)
  - keine Seiteneffekte bzw. explizite globale Effekte (z.B. mittels monadischer Ein/Ausgabe)
  - Berechnung von Ausdrücken in beliebiger (auch paralleler) Ordnung
  - Kontrolle von Seiteneffekten durch das Typsystem

Literaturverzeichnis

[Abelson/Sussman/Sussman 96]
H. Abelson, G.J. Sussman, and J. Sussman. *Struktur und Interpretation von Comptuerprogrammen*. Springer, 1996.

[Antoy 92]
S. Antoy. Definitional Trees. In *Proc. of the 3rd International Conference on Algebraic and Logic Programming*, pp. 143–157. Springer LNCS 632, 1992.

[Antoy/Echahed/Hanus 97]
S. Antoy, R. Echahed, and M. Hanus. Parallel Evaluation Strategies for Functional Logic Languages. In *Proc. of the Fourteenth International Conference on Logic Programming (ICLP'97)*, pp. 138–152. MIT Press, 1997.

[Antoy/Echahed/Hanus 00]
S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, Vol. 47, No. 4, pp. 776–822, 2000.

[Antoy/Hanus 05]
S. Antoy and M. Hanus. Declarative Programming with Function Patterns. In *Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05)*, pp. 6–22. Springer LNCS 3901, 2005.

[Antoy/Hanus 06]
S. Antoy and M. Hanus. Overlapping Rules and Logic Variables in Functional Logic Programs. In *Proceedings of the 22nd International Conference on Logic Programming (ICLP 2006)*, pp. 87–101. Springer LNCS 4079, 2006.

[Antoy/Hanus 09]
S. Antoy and M. Hanus. Set Functions for Functional Logic Programming. In *Proceedings of the 11th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'09)*, pp. 73–82. ACM Press, 2009.

[Antoy/Hanus 10]
S. Antoy and M. Hanus. Functional Logic Programming. *Communications of the ACM*, Vol. 53, No. 4, pp. 74–85, 2010.

[Antoy/Tolmach 99]
S. Antoy and A. Tolmach. Typed Higher-Order Narrowing without Higher-Order

Strategies. In *Proc. 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99)*, pp. 335–352. Springer LNCS 1722, 1999.

[Bry/Schaffert 02]
F. Bry and S. Schaffert. A gentle introduction to Xcerpt, a rule-based query and transformation language for XML. In *Proceedings of the International Workshop on Rule Markup Languages for Business Rules on the Semantic Web (RuleML'02)*, 2002.

[Church 40]
A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, Vol. 5, pp. 56–68, 1940.

[Damas/Milner 82]
L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. 9th Annual Symposium on Principles of Programming Languages*, pp. 207–212, 1982.

[Echahed 88]
R. Echahed. On Completeness of Narrowing Strategies. In *Proc. CAAP'88*, pp. 89–101. Springer LNCS 299, 1988.

[Fay 79]
M.J. Fay. First-Order Unification in an Equational Theory. In *Proc. 4th Workshop on Automated Deduction*, pp. 161–167, Austin (Texas), 1979. Academic Press.

[Fribourg 85]
L. Fribourg. SLOG: A Logic Programming Language Interpreter Based on Clausal Superposition and Rewriting. In *Proc. IEEE Internat. Symposium on Logic Programming*, pp. 172–184, Boston, 1985.

[González-Moreno *et al.* 99]
J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, Vol. 40, pp. 47–87, 1999.

[Hanus 90]
M. Hanus. Compiling Logic Programs with Equality. In *Proc. of the 2nd Int. Workshop on Programming Language Implementation and Logic Programming*, pp. 387–401. Springer LNCS 456, 1990.

[Hanus 91]
M. Hanus. Efficient Implementation of Narrowing and Rewriting. In *Proc. Int. Workshop on Processing Declarative Knowledge*, pp. 344–365. Springer LNAI 567, 1991.

[Hanus 92]
M. Hanus. Incremental Rewriting in Narrowing Derivations. In *Proc. of the 3rd International Conference on Algebraic and Logic Programming*, pp. 228–243. Springer LNCS 632, 1992.

[Hanus 94]
M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, Vol. 19&20, pp. 583–628, 1994.

[Hanus 01]
M. Hanus. High-Level Server Side Web Scripting in Curry. In *Proc. of the Third International Symposium on Practical Aspects of Declarative Languages (PADL'01)*, pp. 76–92. Springer LNCS 1990, 2001.

[Hanus 07]
M. Hanus. Multi-paradigm Declarative Languages. In *Proceedings of the International Conference on Logic Programming (ICLP 2007)*, pp. 45–75. Springer LNCS 4670, 2007.

[Hanus/Huch/Niederau 01]
M. Hanus, F. Huch, and P. Niederau. An Object-Oriented Extension of the Declarative Multi-Paradigm Language Curry. In *Proc. of the 12th International Workshop on Implementation of Functional Languages (IFL 2000)*, pp. 89–106. Springer LNCS 2011, 2001.

[Hanus/Steiner 98]
M. Hanus and F. Steiner. Controlling Search in Declarative Programs. In *Principles of Declarative Programming (Proc. Joint International Symposium PLILP/ALP'98)*, pp. 374–390. Springer LNCS 1490, 1998.

[Hudak 99]
P. Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, 1999.

[Huet/Lévy 79]
G. Huet and J.-J. Lévy. Call by need computations in non-ambiguous linear term rewriting systems. Rapport de Recherche No. 359, INRIA, 1979.

[Huet/Lévy 91]
G. Huet and J.-J. Lévy. Computations in Orthogonal Rewriting Systems. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pp. 395–443. MIT Press, 1991.

[Hughes 90]
J. Hughes. Why Functional Programming Matters. In D.A. Turner, editor, *Research Topics in Functional Programming*, pp. 17–42. Addison Wesley, 1990.

[Hullot 80]
J.-M. Hullot. Canonical Forms and Unification. In *Proc. 5th Conference on Automated Deduction*, pp. 318–334. Springer LNCS 87, 1980.

[Knuth/Bendix 70]
D.E. Knuth and P.B. Bendix. Simple Word Problems in Universal Algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pp. 263–297. Pergamon Press, 1970.

[López-Fraguas/Sánchez-Hernández 99]
F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*, pp. 244–247. Springer LNCS 1631, 1999.

[Martelli/Montanari 82]
A. Martelli and U. Montanari. An Efficient Unification Algorithm. *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 2, pp. 258–282, 1982.

[Milner 78]
R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, Vol. 17, pp. 348–375, 1978.

[Moreno-Navarro/Rodríguez-Artalejo 92]
J.J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic Programming with Functions and Predicates: The Language BABEL. *Journal of Logic Programming*, Vol. 12, pp. 191–223, 1992.

[O'Donnell 77]
M.J. O'Donnell. *Computing in Systems Described by Equations*. Springer LNCS 58, 1977.

[Padawitz 87]
P. Padawitz. Strategy-Controlled Reduction and Narrowing. In *Proc. of the Conference on Rewriting Techniques and Applications*, pp. 242–255. Springer LNCS 256, 1987.

[Robinson 65]
J.A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, Vol. 12, No. 1, pp. 23–41, 1965.

[Smolka 95]
G. Smolka. The Oz Programming Model. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, pp. 324–343. Springer LNCS 1000, 1995.

[Sterling/Shapiro 94]
L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, Cambridge, Massachusetts, 2nd edition, 1994.

[Thompson 96]
S. Thompson. *Haskell - The Craft of Functional Programming.* Addison-Wesley, 1996.

Index